デバッガのためのプログラム実行制御・監視環境の設計と実装

2007年度

孝壽 俊彦

目 次

図 目 次

表 目 次

第1章　序 論

ソフトウェア開発において，開発中のプログラムに不具合が混入することは避けられない．そのため発見された不具合の原因を特定し，修正するデバッグと呼ばれる作業が必要となる．しかしデバッグは，非常に時間のかかる作業となることも多い．そのため多くのソフトウェア開発環境には，デバッグを支援するデバッガ [48] と呼ばれるツールが標準的に含まれている．本章では，まず伝統的なデバッガの概要について述べ，次により高度なデバッガの機能を紹介する．そして，そのような高度な機能を統合したデバッガの構築に必要となる，プログラム実行の制御と監視について述べ，提案の概要について述べる．

1.1　伝統的なデバッガ

デバッガは，伝統的に次の 3 つの基本機能を持つ．

- **ブレークポイント**　プログラムの実行を特定の位置で停止する．
- **ステップ実行**　プログラムを 1 ステップずつ実行する．
- **状態の調査**　変数やスタックなどの状態を調査する．

このような機能を持つデバッガを利用することにより，開発者は，特定の位置でプログラムの実行を停止し，状態を調査することや，1 ステップずつ実行を追跡することができる．

図 1.1 に，伝統的なデバッガのアーキテクチャを示す．ここで，本論文では，デバッガの対象プログラムは C 言語で記述されているものとする．図 1.1 に示したように，伝統的なデバッガでは，OS が提供するデバッグ用の API と，コンパイラが出力するデバッグ情報を利用して，上述のような機能を実現している．

OS のデバッグ用 API の例として，Linux の `ptrace()` システムコール [11] があるが，その機能は非常に低レベルなものである（表 1.1）．そのため，ソースコードでの情報と，実行中のプロセスでの情報との間にギャップが存在する．例えばデバッガは，ソースコードの特定の変数の値を，その型に従った形式で表示することができる．これに対して Linux の `ptrace()` では，実行中のプロセスのメモリの内容を読込むことしかできない．これは他の OS の API でも基本的に同様である．

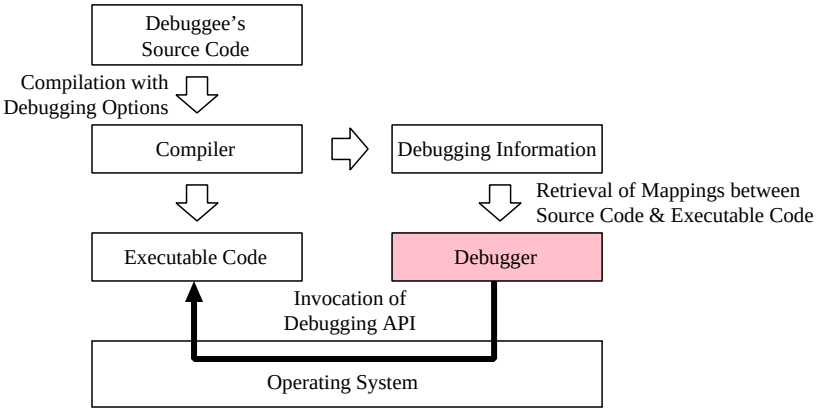


図 1.1 伝統的なデバッガのアーキテクチャ

このようなギャップを埋めるために，伝統的なデバッガでは，コンパイラが出力するデバッグ情報を利用する．図 1.1 に示したように，デバッグ対象プログラム（以下，デバッギと呼ぶ）のコンパイル時にデバッグオプションを指定すると，コンパイラは通常の実行形式に加えて，デバッグ情報を出力するようになる．このデバッグ情報には，ソースコードの行番号や変数と，それらの実行時アドレスを対応付けるための情報や，変数の型情報などが含まれている．デバッガは，この情報に基づいて OS のデバッグ用 API を利用することにより，ソースコードレベルでの機能を提供することが可能となる．なお図 1.1 では，実行コードとデバッグ情報を分離して記載したが，実際には単一の実行形式にまとめられることが多い．

1.2 より高度なデバッガの機能

前節では，伝統的なデバッガが持つ基本的な機能を紹介した．しかし実際には，このような単純な機能だけでは，大規模で複雑なプログラムのデバッグに対して十分な助けにはならない．そのような状況においては，より高度に，プログラム実行の解析を支援できる機能が求められる．ここでは，そのような機能の例として，プログラムスライシングと可逆実行について紹介する[1]．

1.2.1 プログラムスライシング

プログラムのデバッグを行う際，開発者はまず不具合に関連のありそうなコードを特定する必要がある．しかし，プログラムのコード間に複雑な依存関係が存在する場合には，

[1]本研究でこれらの機能に着目した理由は，Agrawal ら [2] により，これらの機能を連係させたデバッグの進め方とシナリオについて報告があり，デバッグに特に有用だと思われたからである．

表 1.1 Linux の ptrace() システムコールの主な機能

コマンド	機能
PTRACE_TRACEME, PTRACE_ATTACH	実行の追跡を開始する
PTRACE_DETACH	実行の追跡を終了する
PTRACE_KILL	プロセスを終了する
PTRACE_CONT, PTRACE_SYSCALL	実行を再開する
PTRACE_SINGLESTEP	機械語命令を 1 命令実行する
PTRACE_PEEKDATA, PTRACE_PEEKTEXT	32bit の値を読み込む
PTRACE_POKEDATA, PTRACE_POKETEXT	32bit の値を書き込む †
PTRACE_GETREGS, PTRACE_GETFPREGS	レジスタの値を読み込む
PTRACE_SETREGS, PTRACE_SETFPREGS	レジスタの値を変更する

† ブレークポイントは，このコマンドを利用し，ブレークポイント命令を直接書き込むことによって実現される．

これは非常に手間のかかる作業となる．伝統的なデバッガでは，この作業を支援するための機能を特に提供していない．

このような場合には，プログラムスライシング [7, 9, 34, 63, 72] と呼ばれる機能が非常に有効である．これはプログラムのコード間の依存関係を解析し，特定の変数の値に影響を及ぼす可能性のあるコードを抽出する機能である．そのため，プログラム中の正しくない値を持つ変数に対して，スライシングを適用することにより，不具合に関連のありそうなコードをしぼりこむことが可能となる．プログラムスライシングでは，プログラムが読み書きした変数などを詳細に監視し，得られた情報を依存関係の解析に役立てることができる．4.1 節で述べるように，これを特に動的スライシングと呼ぶ．

1.2.2 可逆実行

プログラムの不具合は，その原因となるコードが実行されてもすぐには検出されず，後に続くコードの実行中に遅れて検出されることも多い．そのような場合に，開発者は，不具合が検出された箇所からプログラムの実行を逆向きにたどることによって，その原因のある箇所を特定していく場合も多い．しかし伝統的なデバッガでは，プログラムを再実行し，適切な位置（不具合の原因のある箇所と検出された箇所との間）で停止させる作業は，すべて開発者が手動で行う必要がある．この作業は，不具合の原因のある箇所にたどりつくまで反復的に行われるので，非常に手間のかかる作業となる．

このような場合には，可逆実行 [5, 6, 7, 10, 14, 16, 20, 21, 24, 73, 41, 44] と呼ばれる機能が非常に有効である．これは，通常のプログラムの実行方向とは，逆向きの実行を可能にする機能である．4.2 節で述べるように，可逆実行の実現方式には，ロギング方式と再実行方式の 2 種類が存在する．いずれの方式も，通常方向のプログラム実行を監視し，得

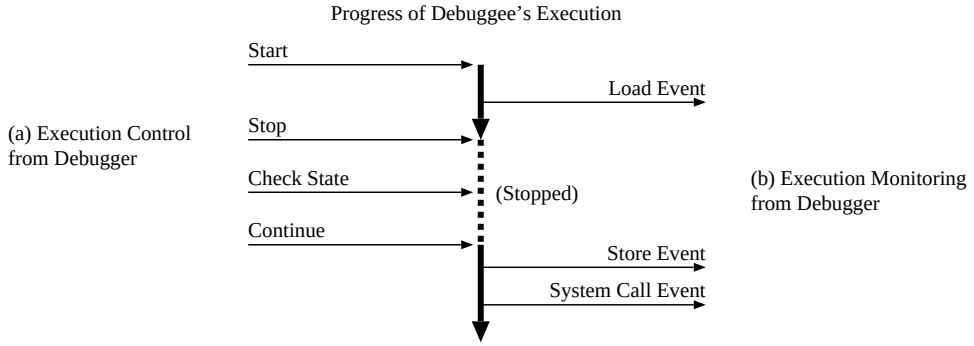


図 1.2 プログラム実行の制御と監視

られた情報を逆向きの実行に活用するものである．

1.3 プログラム実行の制御と監視

前節で述べたような高度な機能を統合したデバッガでは，**プログラム実行の制御と監視**を行う処理が必要となる．ここでプログラム実行の制御と監視について，簡単に定義する（図 1.2）．

- **プログラム実行の制御**　プログラム実行の進捗を管理し，またその状態を調査する処理．本論文では，プログラム実行の開始や終了，停止や継続，状態の取得や修正などの処理を，まとめてプログラム実行の制御と呼ぶ（図 1.2（a））．プログラム実行の制御は，1.1 節で述べたようなデバッガの基本機能を実現するために必要となる．

- **プログラム実行の監視**　プログラムの実行時の振舞いに関する情報を収集する処理．プログラムの実行時の振舞い（メモリの読書き，システムコールの呼出しなど）は，プログラムの実行時に発生するイベントと見なすことが可能である．本論文では，このようなイベントを捕捉し，関連した情報を収集する処理をプログラム実行の監視と呼ぶ（図 1.2（b））．プログラム実行の監視は，前節で述べたような高度な機能を実現するために必要となる．

本研究では，デバッガの開発者に対し，これらの処理を行うための基盤環境を提供する**プログラム実行制御・監視環境（DbgStar）**を提案する．本研究の主な貢献は，次の通りである．

- **仮想マシンによる動的なコード変換のデバッガへの応用**　本研究では，デバッガにおいて，実行の制御と監視を行うための基盤環境が満たすべき要件として，特にオーバー

ヘッド，親和性，柔軟性の 3 つの要件に着目した．しかし次章で述べるように，既存研究で制御・監視のために用いられてきた手法には，これらの要件を同時に満たせるものが存在しなかった．そこで本研究では，**仮想マシンによる動的なコード変換**をデバッガへと応用する．これは，仮想マシン上で，コード変換（SDT：Software Dynamic Translation[52]）を行いながら，プログラムを実行していく手法である．仮想マシンによる動的なコード変換を利用することによる最大の利点は，(1) **実行時に**，(2) **実際に実行した部分のみ**，instrumentation を行えることである．これにより，次の性質を実現できる．

- **低オーバーヘッド**　実際に実行した部分のみ instrumentation を行うため，instrumentation にかかる時間が比較的短い．加えて instrumentation を行ったコードは，CPU 上で直接実行される．そのため全体のオーバーヘッドが比較的低い．
- **高い親和性**　既存のコンパイラが出力した実行形式をそのまま利用するため，デバッギのソースコードに対する変更は基本的に必要としない．また特殊なハードウェアも必要としない．そのため既存の開発環境との親和性が非常に高い．
- **高い柔軟性**　コード変換の詳細を，デバッグを行いながら動的に変更していくことができる．そして，デバッギの実行全体を監視するのではなく，必要な実行区間だけを監視することができる．また監視の粒度も，使用する機能に応じて，自由に調整することができる．そのため柔軟性が非常に高い．

- **デバッガ開発のための基盤環境**　デバッガにおいて，プログラム実行の制御と監視は，前節で述べたプログラムスライシングや可逆実行だけでなく，デバッグに有用な様々な機能の基礎となる処理である．しかしこれらの処理の実装には，大変な労力が必要となる．これは，プログラムの実行には，ハードウェアアーキテクチャ，OS，コンパイラなど，様々なレベルの要素が複雑に関連しているためである．デバッガの開発者は，本研究で提案する DbgStar を利用することにより，開発に必要な労力を大きく軽減できる．

また本研究では，実際に DbgStar を利用して，デバッグに有用な様々な機能を統合したデバッガを構築した．本論文では，特にプログラムスライシングと可逆実行の機能について紹介する．また本論文では，これらの機能を用い，オープンソースプログラム（ProFTPD, GNU Awk，Apache HTTP Server）に含まれていた不具合に対して，デバッグを行うシナリオも紹介する．

1.4 本論文の構成

本論文の構成は，次の通りである．まず第 2 章では，デバッガにおいて，プログラム実行の制御と監視を行うための基盤環境が満たすべき要件を示す．そして，既存研究で制御・監視のために用いられてきた手法を紹介する．第 3 章では，プログラム実行制御・監視環境（DbgStar）を提案する．第 4 章では，DbgStar を利用して，実際に構築したデバッガを紹介する．本章では，特にプログラムスライシングと可逆実行の機能について紹介する．第 5 章では，まず DbgStar のオーバーヘッドについての評価結果を示す．次に，第 4 章で紹介したデバッガのデバッグシナリオを示す．そして，シナリオを通して，DbgStar の親和性と柔軟性について考察を行う．最後に第 6 章で，本論文のまとめと，今後の課題について述べる．

第2章　従来の手法

本章では，まず本研究で着目した要件を示す．次に，既存研究で用いられてきた手法を紹介し，それぞれの問題点を述べる．そして最後に，本章で紹介した従来の手法の問題点を整理する．

2.1　本研究で着目した要件

本研究では，デバッガにおいて，プログラム実行の制御と監視を行うための基盤環境が満たすべき要件として，特に次の 3 つに着目した．

- **オーバーヘッド**　一般的に，実行の監視には，大きなオーバーヘッド（特に時間に関して）が伴われる．このオーバーヘッドがあまりに大きいと，それを利用したデバッガの有用性も損なわれてしまう．このことから，プログラム実行の制御と監視を行うための基盤環境は，監視に伴うオーバーヘッドができるだけ小さいことが望ましい．

- **親和性**　実行の監視のために，特殊なハードウェアが必要であることは，それを利用したデバッガが動作する環境も限られてしまうため望ましくない．同様に，実行の監視のために，デバッギのソースコードに大きな変更が必要となることも望ましくない．このことから，プログラム実行の制御と監視を行うための基盤環境は，既存の開発環境とできるだけ親和性が高いことが望ましい．

- **柔軟性**　デバッガは，使用される機能に応じて，様々な種類の監視を行う必要がある．しかし上述したように，実行の監視には，程度の差はあるが相応のオーバーヘッドが伴われる．そのため，状況に応じて監視の粒度を調整し，必要最小限の監視だけを行うべきである．このことから，プログラム実行の制御と監視を行うための基盤環境は，できるだけ柔軟性が高いことが望ましい．

2.2　既存研究で用いられてきた手法

本節では，既存研究で用いられてきた手法を簡単に紹介し，それぞれの問題点を述べる．また表 2.2 に，前節で述べた 3 つの要件を基に各手法の特徴をまとめたので，あわせて参

照されたい.

2.2.1 トラッピング

トラッピングは，監視が必要な箇所で割込みを発生させ，デバッギの実行を毎回停止させる手法である．そしてデバッギが停止するたびに，実行状態を調査し，監視したい振舞いに関連した情報を収集する．トラッピングでは，割込みを発生させるために，伝統的なデバッガのブレークポイントやステップ実行の機能を利用する．トラッピングのブレークポイントは，デバッガのユーザに対して透過的に設定される．つまり，トラッピングのブレークポイントによってデバッギが停止しても，そのことがユーザに通知されることはない.

トラッピングは，伝統的なデバッガの持つ機能だけで実現できるものである．そのため，一部の伝統的なデバッガには，トラッピングによる監視に基づいた簡単な機能を提供するものもある．そのような例として，GDB[27] のウォッチポイントの機能がある．ウォッチポイントとは，特定の変数の読書きを行うことによって停止する特殊なブレークポイントである（データブレークポイントとも呼ばれる [65]）．GDB では，デバッギのステップ実行を行い，各ステップごとに条件の検査を行うことにより，この機能を実現している.

トラッピングを用いて，より高度な機能を実現したデバッガとして，SPYDER[1, 2] がある．SPYDER は，プログラムスライシングと可逆実行の機能を持つデバッガであり，GCC と GDB を修正して実装されている．SPYDER は，スライスに沿った双方向の実行（通常の実行方向＋逆向きの実行方向）をサポートしている．SPYDER では，スライシングや可逆実行の機能で必要となるメモリの読書きの監視を行うために，デバッギに対し多数のブレークポイントを設定する.

トラッピングには，オーバーヘッドが非常に大きいという問題点がある．これは，ブレークポイントやステップ実行によって停止する度に，割込みが発生するためである．これを確かめるために，本研究では，GDB のウォッチポイントを用いて簡単な実験を行った．実験では，ベンチマーク（ハノイの塔）に対し，決してヒットしないウォッチポイントを 1 つ設定し，実行時間を計測した．その結果，ベンチマークの実行時間が 50,000 倍近くも増加することが観測された.

2.2.2 ページ保護

ページ保護は，多くの OS が持つ仮想メモリの保護機能（`mprotect()` システムコールなど [11]）を利用する手法である．ページ保護では，まず監視が必要なメモリアドレスを含むページに対して，読込みや書込みのアクセスを禁止する．すると，デバッギがそのよ

うなページへアクセスを試みると，例外が発生するようになる．ページ保護では，この例外を検出することにより，メモリの読書きを監視する．ただし例外は，ページ内の任意のアドレスへのアクセスによって発生する．そのため，実際にアクセスされたアドレスが，監視を行いたいアドレスと一致するか検査する必要がある．

ページ保護を利用したデバッガとして，Friday[29] がある．Friday は，分散アプリケーションを対象としたデバッガであり，複数のノードを対象とした分散ブレークポイントや分散ウォッチポイントなどの機能を持つ．Friday は，分散ブレークポイントや分散ウォッチポイントの対象となる各ノードに対し，ローカルなブレークポイントやウォッチポイントを設定する．Friday では，ローカルなウォッチポイントにページ保護を利用している．

ページ保護では，例外の度に割込みが発生する．そのため，保護を行ったページが非常に頻繁にアクセスされる場合には，大きなオーバーヘッドが発生するという問題点がある [18].

2.2.3 ハードウェアによる instrumentation

ハードウェアによる instrumentation は，特殊なハードウェアユニットを用いて，デバッギのコードに対し，監視のためのコードを挿入する（instrumentation と呼ぶ）手法である．例えば Corliss ら [17] は，DISE（Dynamic Instruction Stream Editor）という，プログラム可能なマクロエンジンを提案している．DISE では，DISE エンジンと呼ばれるハードウェアユニットが，プロセッサのフェッチユニットと，実行エンジンの間で動作する．DISE エンジンは，プロセッサがフェッチした命令列に対し，instrumentation を行う役割を持つ．DISE エンジンで行う instrumentation の詳細は，フェッチした命令列に対するマクロ展開のルールとして記述される．DISE のデバッガへの応用として，Corliss らはウォッチポイントを提案している [18].

ハードウェアによる instrumentation は，専用のハードウェアを用いるため高速である反面，そのハードウェアの存在しない環境では利用できない．そのため，既存の開発環境との親和性が低いという問題点がある．

2.2.4 ハードウェアウォッチポイント

Intel x86[31] や PowerPC[28] などのプロセッサには，簡単なウォッチポイントの機能が組み込まれている．例えば Intel x86 は，4 つの Debug Address Register（DR0-DR4）を持つ．これらのレジスタには，監視したいメモリワードのアドレスを設定できる．プロセッサは，設定されたメモリワードへのアクセスを検出すると，割込みを発生させる．割込みが発生する条件は，Debug Control Register（DR7）で調整できる．ハードウェアウォッ

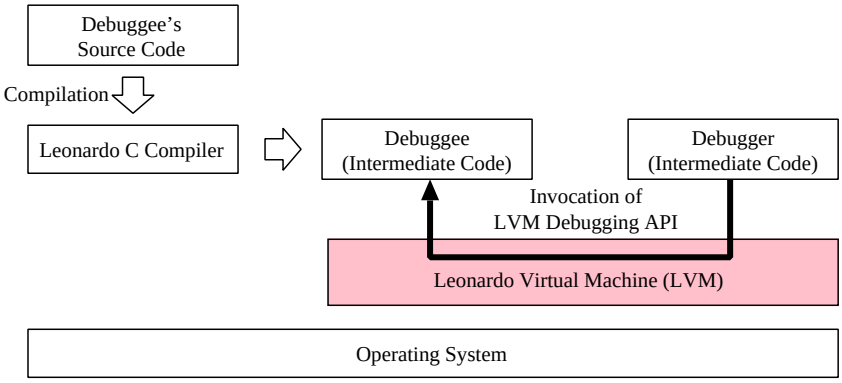


図 2.1 LVM を利用したデバッガ

チポイントは，このようなプロセッサのウォッチポイントの機能を利用し，メモリの読書きを監視する手法である．

2.2.1 節では，GDB のステップ実行を利用したウォッチポイントについて紹介したが，GDB はハードウェアウォッチポイントを利用したウォッチポイントも提供している．GDB では，どちらのウォッチポイントも利用できる場合には，ハードウェアによるものを優先して利用する．

ハードウェアウォッチポイントでは，条件が成立するたびに割込みが発生する．そのため，条件を満たすアクセスが非常に頻繁に行われる場合には，大きなオーバーヘッドが発生するという問題点がある [18]．またハードウェアウォッチポイント自体は，多くのプロセッサでサポートされているものである．そのため，既存の開発環境との親和性に関しては，大きな問題はないと考えられる．ただし，ウォッチポイントの個数に制限があることは，考慮する必要がある．例えば Intel x86 では 4 ワード，PowerPC では 1 ワードまでしか，同時に監視することができない．そのため，ハードウェアウォッチポイントを利用して実現できる機能は非常に限られる．

2.2.5 逐次解釈実行

逐次解釈実行は，デバッギのコードを仮想マシン上で逐次解釈しながら実行し，その振舞いを監視する手法である．逐次解釈実行は，中間コードを逐次解釈するものと，ネイティブコードを逐次解釈するものに分類できる．

LVM（Leonardo Virtual Machine）[21] は，独自の中間言語を逐次解釈実行する仮想マシンである．LVM は，本論文で提案する DbgStar と同様に，プログラム実行の制御と監視を行うための基盤環境を，開発者に提供することを目的として開発されている．図 2.1 に，LVM を利用したデバッガの概要を示す．LVM は，1 つの仮想マシン上で複数のプロ

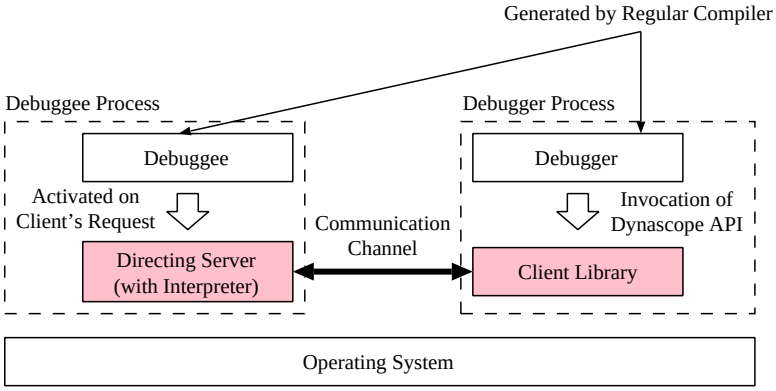


図 2.2 Dynascope を利用したデバッガ

グラムを実行することが可能である．図 2.1 に示したように，デバッガとデバッギも同じ仮想マシン上で実行される．LVM 上で実行されるプログラムは，Leonardo C Compiler を使って，中間コードへコンパイルする必要がある．デバッガは，LVM の提供する API を通して，デバッギの実行の制御と監視を行うことが可能である．また LVM には，可逆実行の機能も組み込まれている．LVM のように独自の中間コードを対象とした逐次解釈実行には，既存のライブラリやシステムコールに関して，互換性の問題を生じる場合がある．例えば 5 章で述べる実験では，ベンチマークを LVM 上で動作させるために，標準入出力やメモリ管理に関連したコードを修正する必要があった．

Dynascope[53, 54] も，やはりプログラム実行の制御と監視を行うための基盤環境である．Dynascope には複数の異なるバージョンが存在するが，ここではネイティブコードを逐次解釈実行するもの [54] について紹介する．図 2.2 に，Dynascope を利用したデバッガの概要を示す．Dynascope は，Client Library と Directing Server の 2 つのコンポーネントを提供する．Dynascope では，デバッガとデバッギを，特殊なコンパイラを使って中間コードへコンパイルする必要はない．代わりに，それぞれに対し，Client Library と Directing Server のモジュールをリンクしておく．図 2.2 に示したように，デバッガは，Client Library を通して Directing Server と通信を行う．Directing Server は，デバッギのプロセス空間で動作するサーバーである．Directing Server は，デバッガのリクエストを処理する必要があるときにだけ起動される．デバッギは，通常は CPU 上で直接実行されている．しかし，デバッガが実行の監視を行う必要がある場合には，Directing Server 内の仮想マシン上で逐次解釈実行される．

逐次解釈実行では，逐次解釈に伴う大きなオーバーヘッドが発生する．5 章で述べるように，本研究では，Stanford Integer Benchmark Suite を利用し，LVM 上で実行した場合の実行時間と，CPU 上で実行した場合の実行時間を比較した．その結果，LVM では，監

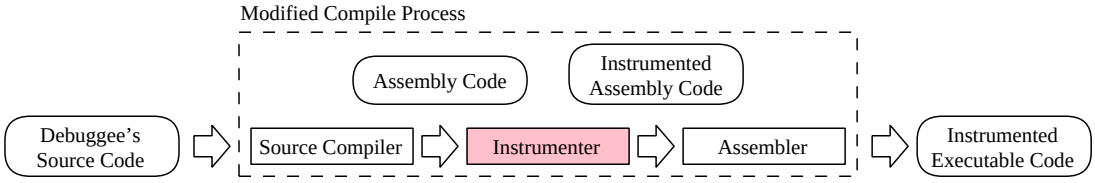


図 2.3 アセンブリコードの instrumentation

視を全く行わなかった場合でも，ベンチマーク自身の逐次解釈のために平均 85.7 倍の実行時間の増加が見られた．また，メモリの書込みと読書きを監視した場合には，それぞれ平均 257.3 倍と 867.6 倍の実行時間の増加が見られた．

2.2.6 静的な instrumentation

静的な instrumentation は，デバッギの**実行前**に instrumentation を行い，デバッギのコードに対し，監視のためのコードを挿入する手法である．instrumentation の対象コードとしては，次の 4 種類が考えられる．

（1）ソースコード

（2）コンパイラ内部の中間表現

（3）アセンブリコード

（4）ネイティブコード

図 2.3 に，(3）の場合の instrumentation を示す．図 2.3 に示したように，アセンブリコードを対象とする場合には，コンパイラのソースコンパイル処理とアセンブル処理の間に，instrumentation 処理が追加される．この場合の Instrumenter の入力は，デバッギのアセンブリコードとなる．また出力は，入力を修正し監視のためのコードを挿入したアセンブリコードとなる．

Wahbe らは，静的な instrumentation を利用したウォッチポイントの機能について研究を行っている [65, 66]．Wahbe らのウォッチポイントは，デバッギのアセンブリコードに対し instrumentation を行い，メモリの書込みを監視することによって実現されている．Wahbe らは，ウォッチポイントのための効率的なデータ構造や，instrumentation 時に解析を行い，不要な監視を省略する手法なども提案している [66]．

丸山らは，静的な instrumentation を利用した可逆実行の機能について研究を行っている [41, 73]．丸山らは，4.2 節で述べる再実行方式の可逆実行を採用しており，プログラムの目標停止地点のタイムスタンプを得るために，静的な instrumentation を利用している．

丸山らは，初期の研究 [73] では，実装の容易さからデバッギのアセンブリコードに対する instrumentation を行っている．これに対し，後の研究 [41] では，（コンパイラがサポートしている限り）特定のアーキテクチャに依存しない，コンパイラ内部の中間表現に対する instrumentation を選択している．丸山らは，幅広いアーキテクチャをサポートしている GNU C Compiler を修正し，その内部表現である RTL（Register Transfer Language）に対する instrumentation を行った．

Chen らも，静的な instrumentation を利用した可逆実行の機能について研究を行っている [14]．Chen らは，4.2 節で述べるロギング方式の可逆実行を採用している．Chen らはデバッギのアセンブリコードに対し instrumentation を行い，レジスタやメモリの書込みなどを監視することにより，プログラムの実行状態の履歴を保存している．

静的な instrumentation は，実行時のオーバーヘッドが比較的小さい手法である．これは，デバッギのコードと監視のためのコードが，共に CPU 上で直接実行されるためである．5 章で述べるように，本研究では，Stanford Integer Benchmark Suite を利用し，静的な instrumentation を行った場合の実行時間と，CPU 上で実行した場合の実行時間を比較した．instrumentation には，ATOM[55] の Intel x86 対応版である，Intel ATOM（Analysis Tools for Object Modification）[30] を用いた．ATOM は，ネイティブコードに対し，静的な instrumentation を行うためのツールである．その結果，メモリの書込みと読書きを監視した場合には，それぞれ平均 7.9 倍と平均 29.1 倍の実行時間の増加が見られた．これは必ずしも高速であるとは言えないが，前節の逐次解釈実行の手法よりは，オーバーヘッドがはるかに小さいことがわかる．また監視を全く行わない場合には，instrumentation も必要ないため，オーバーヘッドが発生しない．

しかし静的な instrumentation には，柔軟性が低いという問題点がある．デバッガが監視する必要のあるデバッギの振舞いは，デバッガのユーザが使用する機能に依存する．そのため，デバッギの実行中であっても，監視する必要のある振舞いは大きく変化する可能性がある．しかし静的な instrumentation では，デバッギの実行前に instrumentation を行う．そのためデバッギの実行中に，そのような変化に対応して，監視コードの追加・削除を行うことはできない．

2.2.7 動的なコードパッチ

動的なコードパッチは，実行中のデバッギプロセスのコードを書き換え，動的な instrumentation を行う手法である．動的なコードパッチにおける instrumentation は，デバッギプロセスのコードに対し，コードパッチを適用する形式で行われる．図 2.4 に，DynInst[13] によるコードパッチの概要を示す．DynInst は，動的なコードパッチのための機能を提供するライブラリであり，Paradyn Project[43] によって開発されている．図 2.4（1）に示した

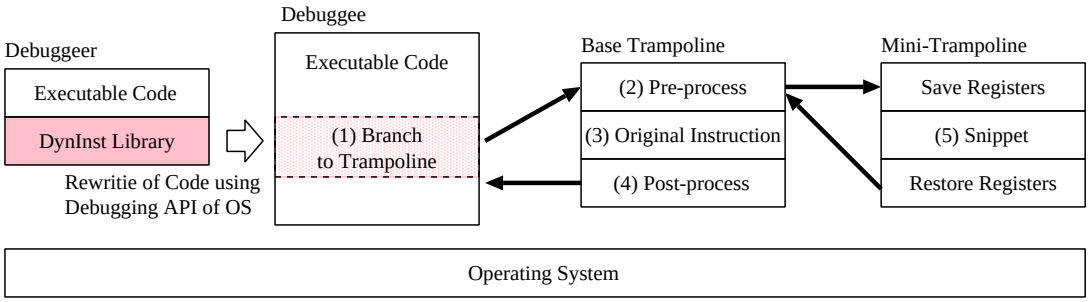


図 2.4 DynInst による動的なコードパッチ

ように，DynInst は，デバッギプロセスの監視が必要な命令を直接書き換え，Trampoline と呼ばれるスタブコードへの分岐命令に変更する．Trampoline には，Base Trampoline と Mini-Trampoline が存在する．Base Trampoline は，まず Mini-Trampoline を呼び出し（図 2.4（2）），次に分岐命令に書換えた元の命令を実行し（3），最後にデバッギの次の命令から実行を再開する役割を持つ（4）．また Mini-Trampoline は，ユーザが用意した監視コード（図 2.4（5）Snippet）を実行する役割を持つ．DynInst では，このようなコードパッチの機能を，OS のデバッグ用 API（1.1 節）を利用して実現している．Buck らは，DynInst のデバッガへの応用例の 1 つとして，条件ブレークポイントを紹介している [13]．また Paradyn Project では，実行中の OS カーネルに対して，DynInst と同様の機能を提供する KernInst[57] の開発も行っている．KernInst の応用例の 1 つとして，柳澤ら [75] が提案した，OS カーネルのデバッグやプロファイリングのためのアスペクト指向システム KLASY が挙げられる．

動的なコードパッチでは，前節の静的な instrumentation の手法とは異なり，**実行時**に instrumentation を行う．そのためデバッギの実行中に，自由にコードパッチを適用すること（監視コードの追加・削除）ができる．そのため，柔軟性が非常に高いと言える．またデバッギのコードと監視のためのコードが，共に CPU 上で直接実行されるため，コードパッチ適用後のオーバーヘッドは比較的小さい．しかしコードパッチの適用自体は，他のプロセス空間のコードを書き換える必要があるため，コストが比較的大きい処理である．また，特に正確な監視を行う必要がある場合には，実行される可能性のあるすべてのコードに対し，コードパッチを適用する必要がある．これには，システムライブラリなどのコードも含まれるため，コードパッチの規模も非常に大きくなりがちである．そのような場合には，コードパッチの適用のために，非常に大きなオーバーヘッドが発生することになる．表 2.1 に，本研究で，Expat XML Parser[58] のサンプルプログラム xmlwf に対し，DynInst を利用してコードパッチを適用した実験の結果を示す．ここでは，xmlwf のメモリの読書きを行うすべての命令に対し，コードパッチを適用した（システムライブラ

表 2.1 DynInst によるコードパッチ

	命令数（K）	時間（分）
DynInst	187	37

リなどのコードも含む）．表 2.1 に示したように，本実験でコードパッチを適用した命令は全部で約 187K 個存在し，適用には約 37 分かかった．

2.3 まとめ

本章で述べてきたように，既存研究で用いられてきた手法には，オーバーヘッド，親和性，柔軟性の 3 つの要件のいずれかにおいて，何らかの問題が存在する（表 2.2）．最後に，それらについて簡単にまとめる．

- **オーバーヘッド** トラッピングの手法は，ブレークポイントやステップ実行のたびに起こる割込みのために，深刻なオーバーヘッドが発生する．また逐次解釈実行の 2 つの手法（中間コード，ネイティブコード）も，静的な instrumentation などと比べると，やはりオーバーヘッドが大きすぎると考えられる．ページ保護，ハードウェアウォッチポイントの 2 つの手法は，割込みを引き起こすアクセスの発生する頻度によって，オーバーヘッドも大きく異なる．そのようなアクセスがほとんどない場合には，オーバーヘッドも小さい．しかし逆に，そのようなアクセスが非常に頻繁に起こる場合には，大きなオーバーヘッドが発生する．実際に，ページ保護を利用して実装されたウォッチポイントが，ステップ実行を利用したものに匹敵するオーバーヘッドを生じるケースも報告されている [18]．動的なコードパッチの手法は，コードパッチの規模によっては，適用のために非常に大きなオーバーヘッドが発生する．

- **親和性** ハードウェアによる instrumentation の手法は，特殊なハードウェアが必要になるため，既存の開発環境との親和性が低い．また逐次解釈実行の手法も，中間コードを対象とする場合には互換性の問題を生じ，デバッギのソースコードに修正が必要となる場合がある．

- **柔軟性** 静的な instrumentation の手法は，デバッギの実行中に監視コードの追加・削除を行うことができないため，柔軟性が低い．これに対し，動的なコードパッチの手法は，そのような操作も可能であるため，非常に柔軟性が高いと考えられる．

表 2.2 既存研究で用いられてきた手法

		オーバーヘッド	親和性	柔軟性
トラッピング		非常に大きい	○	○
ページ保護		アクセス 頻度に依存	○	○
ハードウェア	instrumentation	小さい	要ハードウェア	○
	ウォッチポイント	アクセス 頻度に依存	(要ハードウェア)	○
逐次解釈 実行	中間コード	大きい	互換性に問題 がある場合も	○
	ネイティブコード	大きい	○	○
静的な instrumentation		比較的小さい	○	低い
動的なコードパッチ (instrumentation)		規模に依存	○	○

第3章　提案：プログラム実行制御・監視環境

本章では，前章で示した要件を実現するプログラム実行制御・監視環境（DbgStar）を提案する．

3.1　仮想マシンによる動的なコード変換のデバッガへの応用

仮想マシンによる動的なコード変換とは，仮想マシン上で，コード変換（SDT：Software Dynamic Translation[52]）を行いながら，プログラムを実行していく手法である．仮想マシンによる動的なコード変換は，応用範囲がとても広く，JIT-Compiler[19, 22]，サンドボックス [33, 50]，エラーチェッカ [45, 46]，プロファイラ [38, 45, 46]，動的最適化 [8, 12]，アーキテクチャシミュレータ [15, 68] などに関連した研究が存在する．

図 3.1 に，仮想マシンによる一般的な実行の流れを示す．各ステップは，次の通りである．

（i）レジスタの退避を行い，スタック領域を仮想マシンのものに切り替える．

（ii）次の実行位置からフラグメント（コードの断片）を取り出し，目的に応じたコード変換を行う．

（iii）レジスタとスタック領域を復元する．

（iv）コード変換を行ったフラグメントの先頭にジャンプする．

（v）コード変換を行ったフラグメントを CPU 上で直接実行する．

（vi）フラグメントの実行終了後，(i）に戻る．

本研究では，前章で示した要件を実現するために，仮想マシンによる動的なコード変換をデバッガへと応用する（図 3.2）．

- デバッギの実行形式は，仮想マシン上で実行する．デバッギの実行形式には，既存のコンパイラが出力したものをそのまま用いる（図 3.2（1））．

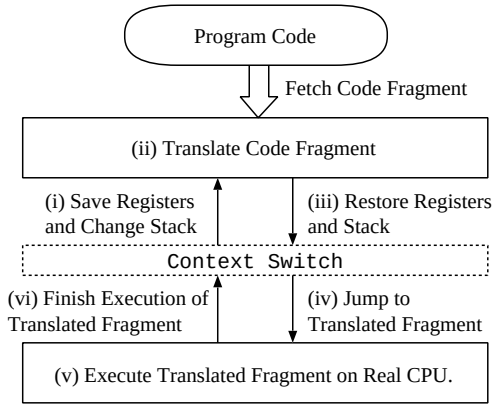


図 3.1 仮想マシンによる実行の流れ

- デバッガは，仮想マシンに対し監視の指示を送りながら，仮想マシンによるデバッギの実行を制御していく（2）.
- 仮想マシンは，デバッガの指示に従ってコード変換（監視に必要な instrumentation）を行いながら，デバッギを実行していく（3）.

これにより，実行の監視に伴うオーバーヘッドを比較的低く抑えることができる．これは，次の 2 つの理由による.

- 動的なコードパッチの手法とは異なり，instrumentation を行うのは，**実行される可能性のあるすべてのコード**ではなく，**実際に実行したコード**のみである．また instrumentation のために，他のプロセス空間のコードを書き換える必要はない．そのため instrumentation にかかる時間が比較的短い.

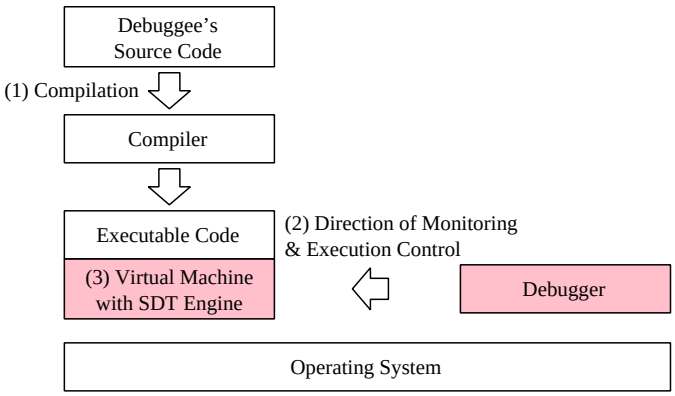


図 3.2 仮想マシンによる動的なコード変換のデバッガへの応用

- 静的な instrumentation や動的なコードパッチの手法と同様に，仮想マシンによって instrumentation が行われたコードは，CPU 上で直接実行される（図 3.1（v））．

また仮想マシンによる動的なコード変換により，既存の開発環境との高い親和性を実現できる．

- ハードウェアによる instrumentation の手法とは異なり，実行の監視に仮想マシンを利用する．そのため，特殊なハードウェアを必要としない．
- 中間コードの逐次解釈実行の手法と異なり，既存のコンパイラが出力する実行形式をそのまま用いる．そのため，デバッギのソースコードに対する変更も，基本的に必要としない．

さらに，高い柔軟性も実現可能である．

- 静的な instrumentation の手法とは異なり，デバッギの実行時に instrumentation を行う．そのため，デバッギの実行中に instrumentation をやり直し，監視コードの追加・削除を行うことが容易である．

以上のように，仮想マシンによる動的なコード変換により，前章で示したオーバーヘッド，親和性，柔軟性の 3 つの要件を同時に実現することができる．なお，本研究と同様に，仮想マシンによる動的なコード変換とデバッガを扱った研究として，Kumar らによるものが存在する [35, 36]．ただし Kumar らの研究は，仮想マシン上で実行されているプログラムを，伝統的なデバッガでデバッグできるようにすることを目的としている．そのため，仮想マシン自体の用途は，特に限定していない（サンドボックスや動的最適化など）．これに対し，本研究の目的は，仮想マシンを利用し，より高度なデバッガの機能を実現することであり，Kumar らの研究とは全く異なるものである．

3.2 プログラム実行制御・監視環境（DbgStar）

プログラム実行の制御と監視は，プログラムスライシングや可逆実行だけでなく，デバッグに有用な様々な機能の基礎となる処理である．そこで本研究では，プログラム実行制御・監視環境（DbgStar）を提案する．DbgStar は，そのような機能の開発に，前節で示した機構を利用できるように設計・実装を行った基盤環境である．本節では，まず DbgStar の全体像について述べ，次にその主要コンポーネントである仮想マシンと操作ライブラリについて述べる．

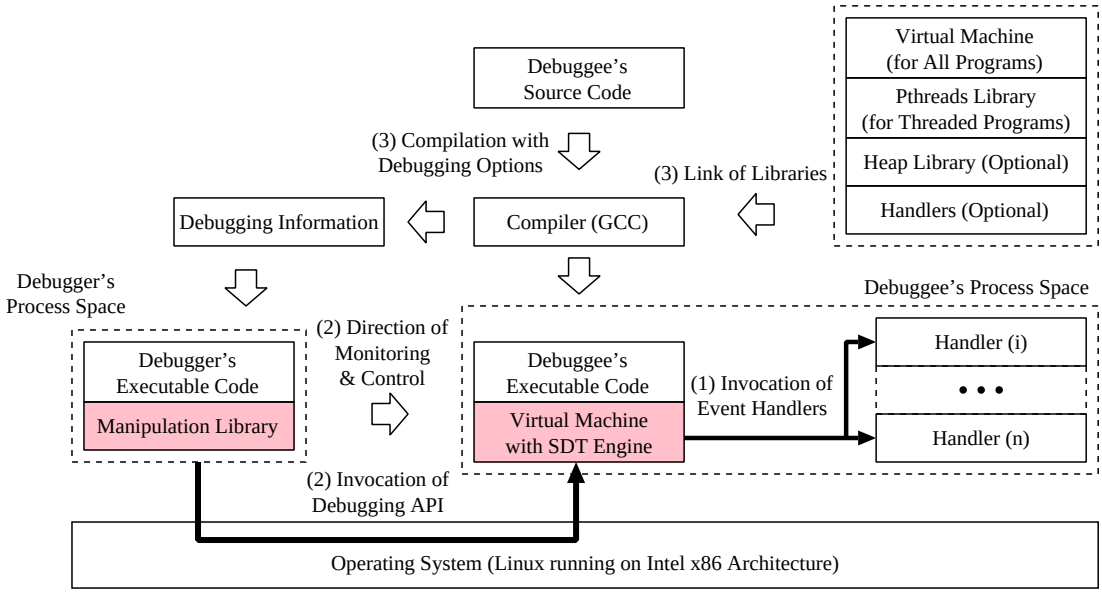


図 3.3 DbgStar の全体像

3.2.1 全体像

図 3.3 に，DbgStar の全体像を示す．DbgStar は，Intel x86 アーキテクチャの Linux 上で動作する．DbgStar は，デバッギとして，C 言語で記述された一般的なユーザアプリケーションを想定している．また DbgStar は，POSIX Threads（Pthreads）API[60] をサポートしている．Pthreads API は，IEEE によって策定されているスレッドプログラミングのための API であり，多くのシステムで利用されているものである．マルチスレッドプログラムの動作は，シングルスレッドプログラムの動作と比べ，非常に複雑である．これは，1 つのプログラム内に複数の実行の流れ（スレッド）が存在し，それらが互いに影響を及ぼしあうためである．また各スレッドの実行される順番が，非決定的であることも，複雑さに拍車をかけている．そのためデバッガには，マルチスレッドプログラムのデバッグを支援するための機能が強く求められる．DbgStar は，そのような機能の開発にも利用することが可能である．

DbgStar は，デバッガの開発者に対し，次の 2 つのコンポーネントを提供する．

- **仮想マシン** 仮想マシンは，デバッギを実行しながら，デバッガの興味のある振舞い（イベント）を見張る役割を持つ．仮想マシンは，デバッギのコード変換時に，そのようなイベントを発生するコードに対し instrumentation を行う．instrumentation されたコードでは，イベント発生時に，デバッガによって登録されたハンドラを呼び出す（図 3.3（1））．デバッガは，このハンドラを通して，イベントに関連した情報を収集する．

- **操作ライブラリ（Manipulation Library）** デバッガに対し，デバッギの実行の制御と監視を行うための API を提供する役割を持つ（図 3.3（2））．操作ライブラリは，内部で OS のデバッグ用 API を利用し，仮想マシンの操作を行う．しかしその詳細は，可能な限り操作ライブラリ内で隠蔽される．

3.2.1.1 デバッギの構築

図 3.3（3）に示したように，デバッギのソースコードは，既存のコンパイラ（GCC）を利用してコンパイルを行う．伝統的なデバッガと同様に，ソースコードレベルでデバッグを行うために，コンパイル時にデバッグ情報を出力する必要がある．またデバッギには，いくつかモジュールをリンクする必要がある．必ずリンクする必要があるものは，仮想マシンである．またマルチスレッドプログラムでは，ダミーの Pthreads ライブラリをリンクする必要がある．このライブラリは，システムの Pthreads ライブラリの利用を防ぐために存在する．なお，スレッド機構の詳細については，3.3.2.3 節で述べる．オプションで，システムのヒープライブラリの代わりに，DbgStar が提供するヒープライブラリをリンクすることもできる．DbgStar が提供するヒープライブラリを用いると，メモリ領域の確保や解放など，ヒープ処理関数に関連したイベントが発生するようになる．デバッガのハンドラに関しては，予めデバッギにリンクしておくことも，操作ライブラリを用い，デバッギの実行中に動的にリンクすることも可能である．

なお，デバッギと上述のモジュールをリンクするためには，デバッギの `configure` スクリプトへのオプションの指定や，`Makefile` の変更などが必要となる．デバッギの構築方法はそれぞれ異なるため，この作業はデバッガの利用者が自らの手で行う必要がある．ただし，それぞれのライブラリ自体は互換性を重視して開発されているので，デバッギのソースコードへの変更は最小限に抑えられるものと期待される．

3.2.1.2 主な機能

表 3.1 に，操作ライブラリの提供する主な機能を示す．表 3.1 に示したように，操作ライブラリの機能は次の 4 種類に分類できる．

（a）起動と終了 “起動” と “終了” は，DbgStar 下でデバッギを起動・終了するための機能である．また “記録” と “再生” は，デバッギを決定的に実行するための機能である．プログラム実行の非決定性とは，プログラムが実行するたびに異なる振舞いを示す性質を指す．デバッギの実行が非決定的であると，デバッギの不具合が，実行するたびに異なる位置で検出される，もしくは全く検出されないといった状況に陥る可能性がある．このよ

表 3.1 操作ライブラリの提供する主な機能

実行の制御に関する機能		
(a)	起動と終了	デバッギを起動，または終了する．
	記録と再生	デバッギを記録，または再生モードで起動する．
(b)	ブレークポイントの設定	ネイティブコードの命令やソースコードの行に ブレークポイントを設定する．
	プロセスの再開	プロセス全体の実行を再開する．
	カレントスレッドの再開	カレントスレッドの実行を再開する．
	カレントスレッドの ステップ実行	カレントスレッドを，ネイティブコードの 1 命令， もしくはソースコードの 1 行分だけ実行する．
	カレントスレッドの切替え	他のスレッドに実行を切り替える．
(c)	変数情報の取得	ソースコードにおける変数の型情報， メモリアドレス，スコープ情報などを取得する．
	メモリの読書き	メモリの内容の取得や修正を行う．
	レジスタの読書き	カレントスレッドのレジスタの内容の 取得や修正を行う．
	スレッド情報の取得	スレッドや同期オブジェクトの一覧， 現在の状態などに関する情報を取得する．
実行の監視に関する機能		
(d)	ハンドラの登録	イベントハンドラを登録する．
	ハンドラの有効化	スレッドに対し，イベントハンドラを有効にする．

うな非決定性が存在すると，デバッグ効率は著しく低下する．DbgStar では，記録モードでデバッギを起動すると，そのような非決定性を生み出す要因をファイルに記録する．そして再生モードでデバッギを起動すると，記録した要因を再生し，全く同じ振舞いを再現することができる．

（b）実行の進捗管理　デバッギの実行の進捗を管理するための機能である．これらの機能は，ネイティブコードレベルとソースコードレベルの，両方のレベルで利用することが可能である．“ブレークポイントの設定” では，特にスレッドの識別は行わず，すべてのスレッドに対して有効なブレークポイントを設定する．個々のスレッドに対するブレークポイントは，その他のスレッドが停止しても単に無視することにより，容易に実現可能である．“プロセスの再開” では，プロセス中のすべてのスレッドの実行が再開される．これに対して，“カレントスレッドの再開” や “カレントスレッドのステップ実行” では，カレントスレッドだけが実行される．またカレントスレッドは，“カレントスレッドの切替え” で，他の実行可能状態のスレッドに切り替えることができる．

（c）実行状態の取得　デバッギの現在の実行状態を取得するための機能である．“変数情報の取得” では，コンパイラが出力したデバッグ情報を基にして，変数の型情報，対応するメモリアドレス，スコープ情報などを取得できる．また “メモリの読書き” では，指定されたメモリアドレスの読書きができる．“変数情報の取得” と “メモリの読書き” を組み合わせることにより，ソースコードにおける変数の値などを調査することが可能である．“レジスタの読書き” では，カレントスレッドのレジスタを読み書きすることができる．ただし，他のスレッドのレジスタを読み書きすることはできない．スタックのバックトレース [48] などが必要となる場合には，“スレッド情報の取得” を利用できる．“スレッド情報の取得” では，デバッギのスレッドや同期オブジェクトに関する情報を取得できる．同期オブジェクトとは，スレッド間で同期を取るために利用されるオブジェクトである．DbgStar では，Pthreads で提供されている Mutex，Reader/Writer Lock，Condition Variable の 3 種類の同期オブジェクトが利用可能である．

（d）監視の指示　仮想マシンに，監視の指示を行うための機能である．前述したように，仮想マシンは，デバッガの興味のある振舞い（イベント）を捕捉すると，デバッガによって登録されたハンドラを呼び出す（図 3.3（1））．“ハンドラの登録” では，監視したいイベントと呼び出したいハンドラの組合わせを，仮想マシンに登録することができる．そして登録したイベントハンドラは，“ハンドラの有効化” で，スレッドごとに有効/無効を切り替えることができる．

ハンドラは，実際には C 言語で記述された関数である．ハンドラには，引数としてイ

ベントに関連した情報が渡される．表 3.2 に，DbgStar で監視可能な主なイベントを示す．例えば LINE と PROC は，それぞれソースコードにおける行や関数の先頭に相当する命令の実行直前に，登録されたハンドラが呼び出されるイベントである．LINE と PROC のハンドラには，引数として命令のアドレスが渡される．またイベントによっては，追加の条件を指定できるものもある．例えば LOAD や STORE では，イベントを特定の範囲のメモリ領域に対するものに限定できる．スレッドや同期オブジェクトに関するイベントは，同名の Pthreads 関数 [60] の機能を反映したものである．さらに DbgStar では，独自のイベント（カスタムイベント）を定義することも可能である．カスタムイベントに対しては，他の組込みイベントと全く同様に，ハンドラを登録することができる．3.2.1.1 節で述べたヒープ処理関数に関連したイベントも，カスタムイベントを利用して実現されている．

なお図 3.3 に示したように，DbgStar では，デバッガとデバッギが別々のプロセスとして実行される．そのため，デバッガはハンドラと何らかの通信を行い，ハンドラが収集した情報をデバッギのプロセス空間から取り出す必要がある．DbgStar では，ハンドラを含むモジュールに対しても，表 3.1（c）の機能を利用することが可能である．そのためデバッガは，ハンドラを含むモジュールのデータ構造にアクセスし，収集した情報を自由に取り出して利用することができる．

3.3 仮想マシン

DbgStar の仮想マシンは，i386 および i486 互換命令セットの中で，C 言語のコンパイラによって出力される一般的な命令を解釈，実行することができる．i386 および i486 互換命令セットは，現在の Intel x86 アーキテクチャの基本となっている命令セットである．そのため GCC[26] などのコンパイラでは，コマンドラインオプションなどを通して，これらの命令セットだけを出力するように指定可能である．

なお，インラインアセンブラなどの機能を利用し，DbgStar の仮想マシンが解釈できない命令を直接記述しているプログラムに対しては，DbgStar を利用することができない．また，実行時にコードの修正や生成を行うようなプログラムに対しても，DbgStar を利用することができない．しかし，これらのプログラムで用いられている手法は，DbgStar が想定しているような一般的なユーザアプリケーションでは，頻繁には利用されないものである．そのため，実用上は大きな問題にならないと考えられる．

表 3.2 監視可能な主なイベント

基本イベント	
LINE，PROC	ソースコードにおける行や関数の実行
INST	ネイティブコードの命令の実行
LOAD	メモリの内容の読込み
STORE	メモリの内容の上書き
CALL，RET	関数の呼出しと復帰
BRANCH	実行の分岐
SYSCALL	システムコールの呼出し
スレッドに関するイベント	
CREATE，EXIT	スレッドの作成と終了
JOIN	スレッドの合流
CANCEL	スレッドの実行のキャンセル
RELEASE	スレッドの資源の解放
SWITCH	コンテキスト・スイッチ
Mutex に関するイベント	
INIT，DESTROY	初期化と破棄
LOCK	ロックの獲得
UNLOCK	ロックの解放
Reader/Writer Lock に関するイベント	
INIT，DESTROY	初期化と破棄
RDLOCK	読込み用のロックの獲得
WRLOCK	書込み用のロックの獲得
UNLOCK	ロックの解放
Condition Variable に関するイベント	
INIT，DESTROY	初期化と破棄
WAIT	待機状態
BROADCAST	シグナルのブロードキャスト
SIGNAL	シグナルの送信

3.3.1 コード変換

仮想マシンは，基本的に図 3.1 のステップに従い，コード変換を行いながらデバッギを実行していく．本節では，特に図 3.1 のステップ（ii）で行うコード変換の詳細について述べる．

DbgStar の仮想マシンにおいて，1 度にコード変換を行うフラグメントの単位は，次の実行位置にある命令から，最初に現れる分岐命令までである．またコード変換されたフラグメントは，単一の入口/出口を持つものとした．これは，将来的に，フラグメント内での最適化を行うことを考慮したためである．

図 3.4 に，DbgStar の仮想マシンが行うコード変換の例を示す．これは，メモリの読込みの監視を行う例である．仮想マシンが行うコード変換は，基本的に次の 2 種類に分類できる．

- **監視コードの挿入** 監視を行う必要がある命令の直前に，ハンドラを呼び出す監視コードを挿入する．図 3.4 の例では，“`mov eax,[ecx]`” がメモリの読込みを行う命令であるため，その直前に監視コードが挿入されている．監視コードでは，まず（1）レジスタの保存とスタックの切替えを行う．次に（2）イベントに関連した情報を引数として，ハンドラ（`load()`）を呼び出す．ハンドラに渡される引数は，先頭から命令アドレス（`pc`），読込みアドレス（`addr`），読込みサイズ（`size`）である．最後に（3）レジスタとスタックを復元し，(4）元の命令を実行する．

- **分岐先の変更** 分岐命令を，本来の分岐先には分岐せずに，仮想マシンに制御を戻すようにコード変換する．そして仮想マシンには，本来の分岐先が次の実行位置（コード変換の開始位置）として渡されるようにする．これは，図 3.1 のステップ（vi）に当たる．図 3.4 の “`jmp label`” が，このコード変換の例である．また関数の呼出しと復帰（`call` 命令と `ret` 命令）に関しては，デバッギのスタックに対し本来のリターンアドレスを直接 `push`，`pop` し，次に `jmp` 命令を用いて仮想マシンへ制御を戻すようにコード変換する．仮想マシンにも，本来の呼出し先，もしくは復帰先を次の実行位置として渡す．これにより，元の命令の動作をエミュレートすることが可能である．

コード変換したフラグメントに対し，仮想マシンはキャッシングとリンキング [51] を行う[1]．キャッシングとは，一度コード変換したフラグメントを，仮想マシン内のキャッシュに登録する手法である．これにより，同一のフラグメントの再変換を避けることができる．またリンキングとは，キャッシュ内のフラグメントの間を，分岐命令で直接結んでしまう手

[1] これらは広く利用されている手法であるが，操作ライブラリによる実行制御（3.4 節）に深くかかわるため，ここで詳しく述べる．

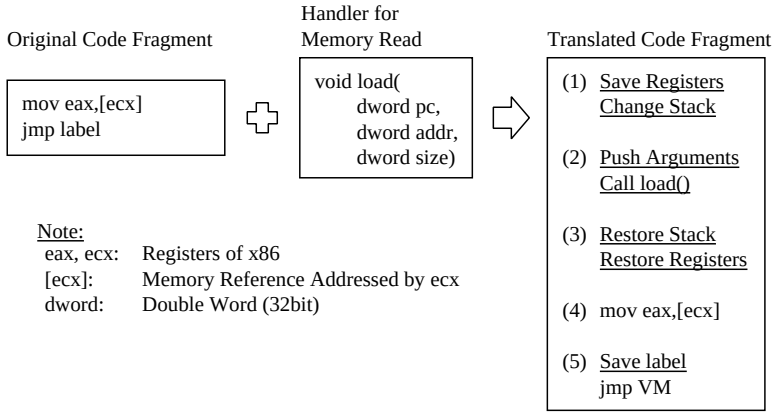


図 3.4 コード変換の例

法のことである．図 3.5 に，リンキングの例を示す．図 3.5（a）に示したように，Fragment 1 と Fragment 2 は，それぞれ分岐命令（Branch）の分岐元と分岐先の関係にある．図 3.5（b）に，Fragment 1 と Fragment 2 が共にコード変換された後の状態を示す．図 3.5（b）では，Fragment 1 の実行後に仮想マシンへ分岐し，仮想マシンから Fragment 2 に分岐している．しかしこの場合には，既に Fragment 2 はコード変換されているため，仮想マシンを経由することは無駄である．そのため，仮想マシンはコード変換した Fragment 1 を書換え，Fragment 2 に直接分岐させる（図 3.5（c））．これにより，図 3.1 のステップ（vi），(i)，(ii)，(iii)，(iv）を省略することができる．

ただし，フラグメントが間接分岐命令（関数の復帰など）で終了している場合には，リンキングを行うことができない．これは，分岐先が実行時まで決定されないためである．そのため Scott ら [51] と同様に，間接分岐命令のための，アセンブリコードで記述した小さなキャッシュを追加した．間接分岐命令では，まずこのキャッシュをチェックする（図

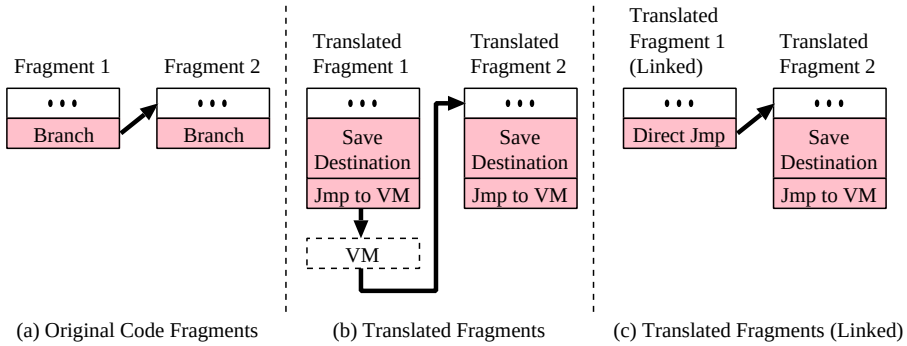


図 3.5 リンキング

3.1 のステップ（v）直後）．そして，分岐先に対応する変換済みフラグメントが見つかった場合には，そのフラグメントへ分岐する．逆に見つからなかった場合にだけ，図 3.1 のステップ（vi），(i)，(ii)，(iii)，(iv）を実行する（本来のキャッシュのチェックは，ステップ（ii）で行われる）．

なお，仮想マシンは，コード変換と同時に**コード変換情報**も生成する．コード変換情報は，仮想マシンによるデバッギの実行を制御するために，操作ライブラリによって利用される情報である．コード変換情報の定義については，付録 A で述べる．またその利用方法については，3.4 節で述べる．

3.3.2 その他の処理

前節で述べたコード変換を行うことにより，デバッギのコードの大部分を実行し，また監視を行うことができる．しかし，システムコール，シグナル，スレッドに関しては，特別な処理が必要となった．

3.3.2.1 システムコール

システムコール命令は，指定されたシステムコール番号に基づいて，様々な種類の処理を行う．仮想マシンでも，それぞれのシステムコール番号に対応した処理が必要である．そこで仮想マシンは，システムコール命令を，仮想マシン内部のシステムコールハンドラの呼出しに置換する．システムコールハンドラでは，システムコール番号に応じて適切なイベント（LOAD，STORE（引数がメモリを参照する場合）や，SYSCALL など（表 3.2））を発生させ，実際にシステムコールを実行する．ただし，システムコールがプロセスの実行をブロックする場合は，システムコールを保留し，他のスレッドに実行を切り替える．現在の実装では，頻繁に利用されるものを中心に，約 70 種類程のシステムコールをサポートしている[2]．その他の多くのシステムコールに関しては，基本的にそのまま実行するため，適切なイベントが発生しないなどの問題を生じる可能性がある．また一部のシステムコールに関しては，DbgStar での実行そのものを禁止している．例えば，DbgStar では `fork()` の実行を禁止している．これは，操作ライブラリ側で必要となる処理が未実装であることなどが原因である．これらのシステムコールへの対応は，今後の課題である．

[2] 5 章で紹介する，ProFTPD，GNU Awk，Apache HTTP Server のデバッグシナリオにおいて利用されていたシステムコールは，すべてサポートしている．

3.3.2.2 シグナル

シグナルには，同期シグナルと非同期シグナルの 2 種類が存在する．同期シグナルは，主にコード実行時のエラーによって発生するものである（不正なメモリの参照など）．これに対し，非同期シグナルは，主に外部から送られてくるものである（キーボードによる割込みなど）．

DbgStar では，同期シグナルは，本来のデバッギの実行とは異なるタイミングで発生する可能性がある．例えば，仮想マシンの実行中に，本来とは異なる形でデバッギのエラーが露呈することも考えられる．また同期シグナルは，主にエラーの発生を示すものであるため，特にハンドラが登録されていないアプリケーションや，登録されていても，エラーログを出力して終了するだけのアプリケーションも多い．そのため DbgStar では，同期シグナルに対するシグナルハンドラの実行を禁止している．同期シグナルの発生時に，実行状態を調査し，その原因を探ることは可能である．

非同期シグナルは，基本的に外部から送られて来るものであるため，ある程度配送を遅延することができる．そのため仮想マシンは，自身に都合の良いタイミングでシグナルハンドラを呼び出す．これは，次のステップで行う．

(1) デバッギによってシグナルハンドラが登録されると，仮想マシンはシステムにダミーのシグナルハンドラを登録する．これにより，実際にシグナルが発生すると，ダミーのシグナルハンドラが呼び出されるようになる．

(2) ダミーのシグナルハンドラでは，シグナルが発生したことを記録しておく．

(3) 仮想マシンは，定期的に記録を調査し，もしシグナルが発生していれば本来のシグナルハンドラを呼び出す．本来のシグナルハンドラは，仮想マシン上でコード変換を行いながら実行される．

DbgStar では，シグナルに関連したシステムコールやライブラリ関数を置換することによって，このような非同期シグナルの遅延配送を実現している．

3.3.2.3 スレッド機構

スレッドの実行制御と監視を行うためには，スレッド機構に対する操作が必要となる．しかし OS が提供するスレッド機構は，そのためのサポートが非常に限定的であった．そこで本研究では，OS が提供するスレッド機構の代わりに，ユーザ空間スレッド機構 [47] を利用することにした．ユーザ空間スレッド機構とは，スレッドの存在を OS が一切関知せず，純粋にユーザ空間内で実装されているスレッド機構である．そのため，拡張することも容易であり，非常に都合が良い．本研究では，GNU Portable Threads（Pth）[23] を

拡張し，仮想マシンのスレッド機構として組み込んだ．Pth は，ユーザ空間スレッド機構を提供するライブラリであり，オープンソースとして公開されている．Pth は，移植性を重視して開発されており，Linux の `clone()` のようなシステムコールも利用しない．Pth を選択した理由は，次の通りである．

- ソースコードが比較的小規模である．
- Pthreads API のエミュレーションレイヤが存在する．

ここでは，仮想マシンに Pth を組み込むために，仮想マシンと Pth のそれぞれに対して行った拡張について述べる．

図 3.6 に，仮想マシンの SDT（Software Dynamic Translation）エンジン，Pth のスケジューラ，Pth の API 間での実行の遷移を示す．ここで SDT エンジンとは，仮想マシンにおいて，コード変換を行いながらデバッギを実行していく（図 3.1 の処理）ユニットを指すものとする．大まかな実行の遷移は，次のようになる．

- スケジューラは，次に実行するスレッドを選択し，SDT エンジンに実行を切り替える（図 3.6（1））．ただし，Pthreads API の呼出しによってスケジューラに遷移していた場合は，呼出し元への復帰を通して SDT エンジンに実行を切り替える（1’）．
- SDT エンジンは，コード変換を行いながら，スレッドを実行していく（2）．
- SDT エンジンは，Pthreads API の呼出しを検出すると，Pth のエミュレーションレイヤの呼出しに置換する（3）．
- エミュレーションレイヤは，Pth API を用い，呼び出された Pthreads API をエミュレートする（4）．
- Pth API は，実行中のスレッドが実行権を放棄するか待機状態に入ると，スケジューラに実行を切り替える（5）．

仮想マシンの管理するデータ領域の中で，レジスタの退避領域やイベントハンドラの管理領域などは，スレッドごとに用意する必要がある．そのため，そのようなデータ領域は，スレッド管理領域として分離した（図 3.6（6））．また Pth のスレッド作成関数にも修正を行い，スレッド管理領域を割り当て初期化を行い，新規作成スレッドは SDT エンジン上で実行を開始するようにした．

また Pth のスケジューラは，SDT エンジンに切り替わる際に，選択したスレッドの管理領域を，SDT エンジンの参照する領域にコピーするように変更した（図 3.6（7））．逆にスケジューラに戻ってきた際には，SDT エンジンの参照した領域を，再びスレッド管理領域にコピーする（図 3.6（8））．

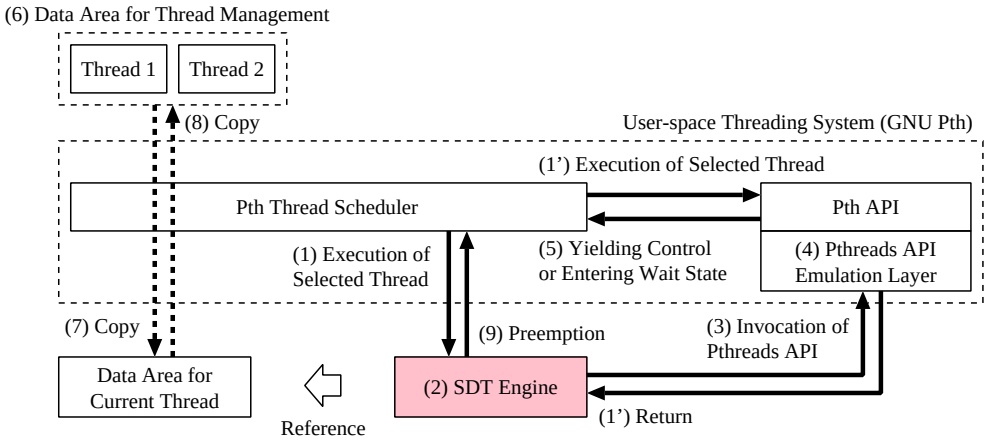


図 3.6 実行の遷移

そして SDT エンジンを修正し，Pthreads API の呼出し[3]を，Pth のエミュレーションレイヤの呼出しに置換する機構を追加した（図 3.6（3））．また Pth のスケジューラと API を修正し，スレッドに関連したイベント（表 3.2）を検出する機構を追加した．データ構造も拡張し，スレッドや同期オブジェクトに関する情報を，操作ライブラリから解析しやすいようにした．

Pth は実装がシンプルである反面，ノンプリエンプティブなスケジュールしか行うことができない．すなわち，各スレッドが自発的に実行権を放棄した場合か，Pth API [4]を呼び出して待機状態に入った場合にだけ，スケジューリングが行われる（図 3.6（5））．そこで SDT エンジンを修正し，各スレッドのコードをある程度実行すると，強制的にスケジューラに実行を切り替えるようにした（図 3.6（9））．DbgStar では，各スレッドを一度に実行する量を指定するために，SDT エンジンが実行するフラグメントの個数を用いる．

3.3.3 実行の記録・再生

3.2.1.2 節で述べた通り，デバッギの実行が非決定的であると，デバッグ効率は著しく低下する．そのため，DbgStar の仮想マシンには，実行の非決定性を生み出す要因を記録・再生する機構を実装した．

本研究で実装した記録・再生機構は，Mellor-Crummey ら [42] や Russinovich ら [49] の SIC（Software Instruction Counter）を利用した手法を基にしている．SIC とは，プログラム実行時に，次の条件によってインクリメントされるカウンタである．

(1) 後方アドレスへの分岐命令の実行

[3]実際には，ダミーの Pthreads ライブラリに含まれる，本体が空の関数の呼出しである（3.2.1.1 節）．
[4]一部のシステムコール（`pth_select()`，`pth_read()` など）や，同期オブジェクトの操作など．

(2) 関数の呼出し

そのため，同一の命令が繰り返し実行される場合には，SIC には常にユニークな値が与えられる．SIC の値とハードウェアの提供する PC（Program Counter）の値のペアにより，プログラムの正確な実行位置を識別することが可能となる．Mellor-Crummey らは，非同期イベント（割込みなど）の発生時に SIC と PC を記録し，再実行時に全く同じタイミングでイベントを再生する機構を提案している．また Russinovich らは，その機構をスレッドのスケジューリングにまで拡張している．どちらの場合も，アセンブリコードに対し静的な instrumentation を行うことにより，SIC を実現している．

本研究でも，まず仮想マシンを拡張し SIC を導入した．DbgStar では，基本的に次のタイミングで SIC をインクリメントする．

（i）後方アドレスへの分岐命令

（ii）関数の呼出し

（iii）間接分岐命令

（iv）システムコールの前後

（v）Pthreads 関数の前後

（iii）は，（i）や（ii）との区別が難しいために含めたものである．また（iv）と（v）は，システムコールや Pthreads 関数の実行中に，スレッドのスケジューリングが行われる可能性があるために含めたものである．

また本研究では，実行の非決定性を生み出す要因として，次の 3 つに着目した．

- **システムコール**　多くのシステムコールの実行結果は，外部の状態（ファイルシステム，他のプロセスなど）の影響を受ける．しかし，外部の状態すべてを記録・再生することは，現実的ではない．そのため DbgStar では，システムコールの実行結果の記録・再生を行う．まず記録実行時には，システムコール実行時の SIC と PC，戻り値，メモリの変更結果（例えば，`read()` では読込んだファイル内容）を記録する．そして再生実行時には，実際にはシステムコールを実行せず，記録された戻り値とメモリの変更結果だけを再現する[5]．

- **非同期シグナル**　3.3.2.2 節で述べた通り，仮想マシンは定期的にシグナルの発生記録を調査し，もし発生していればデバッギの登録したシグナルハンドラを呼び出す．DbgStar では，シグナルハンドラの呼出しの記録・再生を行う．まず記録実行時に

[5]ただし，例外として，`brk()` や `mmap()` などは実際に実行する．

は，シグナルハンドラ実行時に SIC と PC，シグナル番号などを記録する．そして再生実行時には，本来はシグナルの発生記録を調査していたタイミングで，SIC と PC を調査する．一致していた場合は，記録された情報に基づいて，シグナルハンドラを呼び出す．

- **スレッド機構** DbgStar では，ユーザ空間スレッド機構を採用しているため，マルチプロセッサシステムであっても，複数のスレッドが同時に実行されることはない．そのため，各スレッドが共有資源にアクセスする順番は，スレッドのスケジューリングによって決定される．そこで DbgStar では，スレッドのスケジューリングの記録・再生を行う．まず記録実行時には，スケジュール発生時に SIC と PC，スケジュール前後のスレッドの識別子などを記録する．そして再生実行時には，スケジューリングが発生する可能性のあるすべてのタイミングで，SIC と PC を調査する．一致していた場合は，スケジューラに実行を切り替える．スケジューラは，記録された情報に基づいて，次に実行するべきスレッドを選択する．

なお，記録・再生実行時には，一部の操作ライブラリの機能が利用できなくなる．例えば，メモリやレジスタの内容の修正（表 3.1）などである．これは，これらの機能が，デバッギの実行時の振舞いを変化させてしまうためである．

3.4 操作ライブラリ

操作ライブラリは，Linux の `ptrace()` システムコール（表 1.1）や `proc` ファイルシステム [11] を用いて，仮想マシンやデバッギの操作を行う．本節では，表 3.1 に示した機能に沿って，操作ライブラリの実装について述べる．

3.4.1 実行の制御に関する機能

表 3.1 に示したように，実行の制御に関する機能は 4 種類に分類できる．

3.4.1.1 起動と終了（表 3.1（a））

操作ライブラリによって起動された直後は，デバッギは CPU 上で直接実行されている．操作ライブラリは，まず ELF ヘッダの e_entry フィールドに設定されたアドレス [37] まで，デバッギを実行する．このアドレスは，システムが最初に制御を渡す，プロセスの開始アドレスである．次に，デバッギや仮想マシンなどのモジュールから，デバッグ情報（1.1 節参照）を読込む．そして最後に，次のステップでデバッギの実行を仮想マシンに切り替える．

(1) 仮想マシンが次の実行位置を格納している変数に，現在の PC の値を設定する．

(2) PC の値を，図 3.1 のステップ（i）を行うコードのアドレスに変更する．

(3) 図 3.1 のステップ（i）から（iv）までを実行する．

また実行の記録・再生を行う場合には，デバッギ起動時に環境変数を設定し，関連した情報（ファイル名など）を仮想マシンに通知する．

3.4.1.2 実行の進捗管理（表 3.1（b））

従来のデバッガとは異なり，操作ライブラリは，仮想マシン上で実行されているデバッギに対し，ブレークポイントの設定やステップ実行を行う．そのためには，仮想マシンが行ったコード変換の詳細に関する情報が必要となる．

そこで本研究では，そのような情報をやり取りするために，**コード変換情報**を定義した（付録 A）．仮想マシンはコード変換時に，コード変換情報を生成する．そして操作ライブラリは，仮想マシンが停止するたびに，仮想マシンのデータ領域から（新たに生成された）コード変換情報を取得する．操作ライブラリは，コード変換前後のコードの対応関係などの情報をコード変換情報から取得し，ブレークポイントの設定やステップ実行を行う．ここでは，これらの機能の機械語コードレベルでの実装について紹介する．これらの機能は，伝統的なデバッガと同様に，コンパイラが出力するデバッグ情報を用い，比較的簡単にソースコードレベルにまで拡張することができる（1.1 節参照）．

ブレークポイントの設定 3.3.1 節で述べたように，仮想マシンがコード変換したフラグメントは単一の入口/出口を持つ．そのため，同じ命令が何度も再変換され，対応するコードが複数の変換済みフラグメントに現れる可能性がある．図 3.7 に，例を示す．まず，図に示されていない Instruction 1 を分岐先とする分岐命令が実行されたとする．すると仮想マシンは，Instruction 1 から始まるフラグメントをコード変換する（図 3.7（b））．同様に，Instruction 2 を分岐先とする分岐命令が実行されたとする．すると仮想マシンは，先程コード変換したフラグメント（b）を再利用するのではなく，Instruction 2 から始まるフラグメントを新たにコード変換する（c）．そのため，Instruction 2，3，4 が 2 度コード変換され，別々の変換済みフラグメント（b），(c）に対応するコードが現れることになる．

このような命令については，対応するすべてのコードに対し，ブレークポイントの設定を行う．コード変換前後のコードの対応関係は，コード変換情報から取得することが可能である．図 3.7 において，Instruction 3 にブレークポイントを設定する場合を考える．Instruction 3 に対応するコードは，2 つの変換済みフラグメント（b），(c）に存在する．またこの例では，Instruction 3 の直前に監視コードが挿入されている．そのため操作ライ

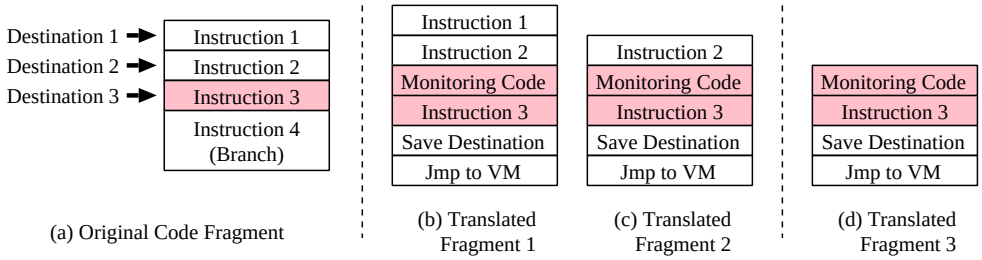


図 3.7 命令の再変換

ブラリは，変換済みフラグメント（b），(c）に挿入された監視コードの先頭にブレークポイントを設定する．

ただし，以上のようにブレークポイントを設定した命令が，再度変換される可能性もある．図 3.7 で，実行を再開した場合を考える．ここで，もし Instruction 3 を分岐先とする分岐命令が実行されると，Instruction 3 から始まるフラグメントが新たにコード変換される（d）．

このような場合には，ブレークポイントの設定は，操作ライブラリではなく仮想マシンが行う．操作ライブラリは，ブレークポイントの設定を行うたびに，関連した情報を仮想マシンのデータ領域に書き込む．仮想マシンは，この情報を基に，コード変換を行うコードにブレークポイントが設定されていないか検査を行う．そして設定されている場合には，変換済みフラグメントにも対応するブレークポイントを設定する．その際に仮想マシンは，コード変換時に設定したブレークポイントの情報を，コード変換情報に出力する．このコード変換情報は，操作ライブラリがブレークポイントの削除を行う際に必要となる．

ステップ実行　仮想マシンは，単一の命令を，複数の命令からなるコードにコード変換する場合がある．操作ライブラリは，そのようなコードをまとめて実行する．図 3.7（b），(c)，(d）では，Instruction 3 と Instruction 4 [6]が，複数の命令からなるコードにコード変換されている．これらの命令をステップ実行する場合には，操作ライブラリは，まず対応するコードの末尾の命令（Instruction 3 や仮想マシンへの分岐命令）に，一時的なブレークポイントを設定する．そしてデバッギを，ブレークポイントまで実行する．最後に，ブレークポイントを削除し，残された 1 命令を実行する．ここで，仮想マシンへの分岐命令を実行した場合には，次の変換済みフラグメントの先頭まで実行を継続させる．

仮想マシンは，単一の命令を複数の命令からなるコードにコード変換する場合に，末尾の命令のアドレスをコード変換情報に出力する．ただし，末尾の命令のアドレスは，変換

[6]分岐先の保存（“Save Destination”）と，仮想マシンへの分岐命令（“Jmp to VM”）にコード変換されている．

済みフラグメントのリンキングによって変更される場合がある（3.3.1 節参照）．図 3.5（b）に示したように，Fragment 1 の分岐命令に対応するコードの末尾の命令は，コード変換直後には仮想マシンへの分岐命令である．しかし仮想マシンがリンキングを行うと，末尾の命令は Fragment 2 への分岐命令となる（図 3.5（c））．そのため仮想マシンは，リンキングを行う際に，末尾の命令に関するコード変換情報を出力しなおす必要がある．

その他の機能　“プロセスの再開” では，ユーザ空間スレッド機構に対する操作を行わず，プロセス全体の実行を再開する．これに対し，“カレントスレッドの再開” と “カレントスレッドのステップ実行” では，まず SDT エンジンを操作し先取り（図 3.6（9））を禁止した上で，プロセスの実行を再開する．ただし，先取りを禁止していても，自発的に実行権を放棄した場合や待機状態に入った場合（図 3.6（5））には，スレッドの切替えが起こる．また再生実行時には，先取りの設定に関わらず，記録実行時と全く同じタイミングでスレッドの切替えが起こる．そのような場合には，スケジューラを操作し，次のスレッドの再開位置でデバッギの実行を停止させる．“カレントスレッドの切替え” では，まず目標のスレッドの識別子をスケジューラのデータ領域に書き込む．次に，デバッギの実行をスケジューラに切り替え，スケジューラに目標のスレッドを選択させる．そして目標のスレッドの再開位置で，デバッギの実行を停止させる．

3.4.1.3　実行状態の取得（表 3.1（c））

“メモリの読書き” では，単純に指定されたメモリアドレスの読書きを行う．これに対し “レジスタの読書き” では，レジスタを直接読み書きするだけではなく，レジスタが仮想マシンのデータ領域に退避されている場合には，その退避領域に対する読書きを行う．また “スレッド情報の取得” では，ユーザ空間スレッド機構から必要なデータを取出し解析を行う．

3.4.2　実行の監視に関する機能（表 3.1（d））

仮想マシンは，**監視テーブル**と呼ぶデータ構造を参照して，デバッギの実行の監視を行う．図 3.8 に，監視テーブルの概要を示す．監視テーブルは，実際には 32bit のフラグである．監視テーブルの各ビットは，イベントハンドラに対応している．各ビットに対応するイベントハンドラは，予め “ハンドラの登録” で登録を行っておく．各ビットには，複数のイベントハンドラを登録することが可能である．デバッガは，提供する機能ごとにイベントハンドラを分類し，異なるビットを割り当てると便利である．

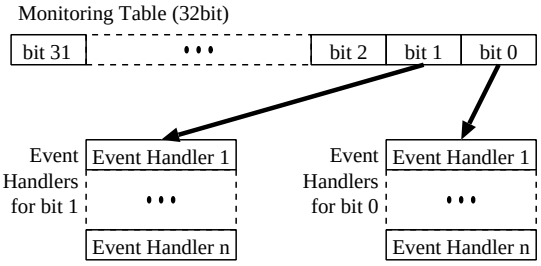


図 3.8 監視テーブル

“ハンドラの有効化” では，監視テーブルのビットを操作し，各ビットに対応するイベントハンドラの有効/無効を切り替える．仮想マシンの監視テーブルは，スレッドごとに個別に用意されている．そのため，イベントハンドラも，スレッドごとに切り替えることが可能である．また監視テーブルが一致するスレッド間では，キャッシュも共有される．監視テーブルを変更した際には，操作ライブラリを利用して，仮想マシンのキャッシュをフラッシュする必要がある．すると操作ライブラリは，内部に保存された情報（コード変換情報，変換済みフラグメントに設定したブレークポイントなど）も，必要に応じ破棄，または更新を行う．

ハンドラの有効化には，2 種類の方法がある．まず特定のスレッドの監視テーブルを，直接操作する方法である．ただしこの方法では，スレッドの個数が多い場合に不便である．そのためもう 1 つの方法として，**スレッドグループ**を利用する方法を用意した．スレッドグループは，本研究で Pthreads API を拡張して導入した概念である．Pthreads API では，スレッドに対して様々な属性を設定することができる [47]．スレッドグループも，その拡張属性として定義した．デバッガは，同一のスレッドグループに属するスレッドに対し，まとめて監視テーブルを設定することが可能である．また各スレッドグループに対し，デフォルトの監視テーブルを設定することも可能である

なお，スレッドグループを活用するためには，デバッギのソースコードを修正し，各スレッドに対しグループを設定するためのコードを挿入する必要がある．しかしこのコードは，DbgStar に情報を与えるためだけのものであり，実行自体に影響を与えるものではない．そのため#ifdef などを用いて，必要な場合にだけ埋め込むことができる．

3.5 制限

DbgStar には，3.3.2.1 節で述べたシステムコールの制限に加え，次の制限が存在する．

- **互換性** DbgStar は，Pthreads API[60] の仕様との完全な互換性は実現していない．例えば，DbgStar では，Pthreads API の仕様に存在するいくつかの属性をサポートし

ていない（mutex を複数のプロセス間で共有するための属性など）．これは，DbgStar のユーザ空間スレッド機構が基にした，GNU Pth の Pthreads API エミュレーションレイヤに起因する制限である，

- **本来の実行との差異** DbgStar でのデバッギの実行は，仮想マシン上で行われるため，本来の実行とは異なった振舞いになる可能性がある．例えば，3.3.2.2 節で述べたように，同期シグナルが本来の実行とは異なるタイミングで発生する可能性がある．しかし，この問題を完全に解決することは非常に難しく，また同様の問題は，2 章で紹介した多くの機構が抱えている．本研究では，少しでもこの問題を緩和するために，デバッギの実行に与える影響が可能な限り少なくなるように，仮想マシンの設計・実装を進めた[7]．

- **不具合の発生** 3.3.2.3 節で述べたように，DbgStar では，SDT エンジンの実行するフラグメントの個数を用いて，先取りの間隔を設定することが可能である．これにより，システム標準のスレッド機構などと比べても，はるかに小さい間隔で先取りを行うことができる．しかし DbgStar では，フラグメントの途中では先取りが起きない．これは，そのような位置で先取りが起きなければ，デバッギの不具合が発生しない場合などに大きな問題となる．実際には，DbgStar を少し拡張することにより，フラグメントの途中で先取りが起きるようにすることは簡単である．しかし，任意の位置でのスケジューリングを許可すると，スケジューリングの発生条件の検査に伴うオーバーヘッドが非常に大きくなるという問題がある，そのため，単純に任意の位置でのスケジューリングを許可するのではなく，不具合が発生しやすい先取りのタイミングなどを研究する必要がある．これは今後の課題である．

- **正当性の保証** DbgStar を利用して構築したデバッガでは，デバッギの正当性を保証することはできない．デバッガは，あくまで発見された不具合の修正を支援するツールであり，プログラムに欠陥がないことを証明できないという限界がある．プログラムの正当性を保証するためには，形式的に記述された仕様に基づいて検証を行う必要がある [74]．

[7]例えば仮想マシンは，ごく一部の関数を除いて，システム標準のライブラリ関数を利用しないようにした．

第4章　デバッガ

前章で提案したDbgStarを利用して，実際に構築したデバッガを紹介する．本デバッガの基本的な利用方法は，伝統的なデバッガと同様である．しかし本デバッガでは，伝統的なデバッガには存在しない様々な拡張機能が利用可能である．例えば，実行の記録・再生，プログラムスライシング，可逆実行などである．

実行の記録・再生は，既に3.3.3節で述べた通り，DbgStarに組み込まれている機能である．本デバッガでは，次の`record`コマンドで実行の記録を行い，`replay`コマンドで再生を行うことができる．

```
record ファイル名 デバッギ デバッギの引数
replay ファイル名
```

ここで“ファイル名”は，非決定性を生み出す要因を記録するファイルの名前である．またデバッガのUIを経由せずに，シェルから直接実行の記録を行うことも可能である．その場合には，デバッガ起動時のコマンドラインに，`record`コマンドと同様の引数を渡す．

以下では，プログラムスライシングと可逆実行の機能について詳しく紹介する．

4.1　プログラムスライシング

本節では，プログラムスライシングの概要と，本研究での実装について述べる．

4.1.1　概要

プログラムスライシングとは，プログラムのコード間の依存関係を解析し，特定の変数の値に影響を及ぼす可能性のあるコードを抽出する機能である [63, 70]．

入力と出力　スライシングへの入力は，Slicing Criterionと呼ばれ，$< p, V >$といった形式で表されることが多い [70]．ここでpとVは，それぞれプログラム内部の位置と変数（の集合）である．またスライシングの出力は，プログラム位置pにおいて，変数Vの値に影響を及ぼす可能性のあるステートメントの集合である．実際に，何をステートメントとするかは，実装により異なる（ソースコードの行，ネイティブコードの命令など）．スライシングの出力は，スライスと呼ばれる．

依存関係　スライスは，ステートメント間の依存関係に基づいて計算される．主要な依存関係として，コントロール依存とデータ依存が存在する [63]．コントロール依存は，条件ステートメントと，条件によって選択されるステートメント間の関係を捉えるものである．構造化されたプログラムにおいては，`if` や `while` などのブロック内部のステートメントが，`if` や `while` 自身にコントロール依存をする[1]．これに対し，データ依存は，ステートメント間の Def-Use 関係を捉えるものである．$def(s)$，$use(s)$ を，それぞれステートメント s で定義，使用される変数の集合とする．すると，ステートメント $s1$ と $s2$ の間に次の関係が成り立つとき，$s2$ は $s1$ にデータ依存をすると定義する．

(1) $\exists x : x \in def(s1) \cap x \in use(s2)$

(2) $s1$ から $s2$ の間に，x を定義するステートメントが存在しないパスが存在する．

静的スライシングと動的スライシング　スライスの計算手法は，静的スライシングと動的スライシングに分類できる．静的スライシング [67] は，**プログラムのあらゆる実行を考慮し**，変数の値に影響を及ぼす可能性のあるすべてのステートメントを抽出する手法である．これに対し，動的スライシング [3, 9, 34, 71, 72] は，**プログラムのある特定の実行だけを考慮し**，変数の値に実際に影響を及ぼしたステートメントだけを抽出する手法である．

Backward Computation と Forward Computation　動的スライシングは，さらに Backward Computation と Forward Computation の 2 種類の手法に分類できる．Backward Computation[3, 71] は，プログラム実行時にトレース情報を保存しておき，その情報を基に DDG（Dynamic Dependence Graph）を構築する手法である．DDG とは，プログラム内部の依存関係を表すグラフである．これに対し，Forward Computation[9, 34, 72] は，プログラムの実行と並行して，プログラム中のすべての変数に対するスライスを計算する手法である．

4.1.2　方針

本研究では，データ依存関係に基づいてスライシングを行う．これは，次の理由による．

- コントロール依存関係も含めると，スライスサイズが非常に大きくなる場合がある．
- 次章で示すように，データ依存関係の解析だけでも多くの不具合に対して有効である．

また本研究では，動的スライシングを行う．静的スライシングでは，プログラムのあらゆる実行を考慮するため，スライスのサイズが大きくなりやすいためである．そして本研究では，Forward Computation を行う．これは，次の理由による．

[1] 正確には，CFG（Control Flow Graph）における Post-Dominance 関係によって定義される．

表 4.1 スライシングで利用される主なイベントハンドラ

イベント	ハンドラ
LOAD	`void __slice_use(dword inst, dword addr, dword size);`
STORE	`void __slice_def(dword inst, dword addr, dword size);`
INST	`void __slice_inst(dword inst, dword load, dword store);`

- Backward Computation では，トレース情報や DDG に大量のメモリが必要となる．
- Forward Computation では，roBDD（Reduced Ordered Binary Decision Diagram）[2]を利用した非常にコンパクトなデータ構造が利用できる [72].

4.1.3 実装

イベントハンドラ イベントハンドラでは，ネイティブコードレベルでスライシングを行う．そしてソースコードへのマッピングは，ユーザがスライスを取得するときに行う．スライシングでは，10 種類以上のイベントハンドラを利用するが，主要な役割を果たすものは，表 4.1 に挙げた 3 つである．`__slice_use()` と `__slice_def()` は，メモリの読書きが行われる際に呼び出されるハンドラである．また`__slice_inst()` は，個々の命令の実行直前に呼び出されるハンドラである．メモリの読書きを行う命令では，まず`__slice_use()` と `__slice_def()` が呼び出され，次に`__slice_inst()` が呼び出される．

図 4.1 に，これらのハンドラの処理を疑似コードで表したものを示す．図 4.1 において，*slice* は現在までに計算されたスライスの集合である．*slice*[*address*] で，個々のアドレスのスライスを表す．本研究では，*slice* のデータ構造として，Zhang ら [72] と同様に roBDD を用いる．実装には，オープンソースの BDD ライブラリである BuDDy[32] を利用した．また *use* と *def* は，現在の命令によって使用，または定義されるアドレス（の集合）に関する情報を保持する．*use* は，使用されるアドレスのスライスを保持する．*def* は，定義されるアドレスそのものを保持する．

`__slice_use()` では，読込みを行う領域の各アドレスについてスライスを取得し，それを *use* に追加していく（図 4.1（1））．同様に`__slice_def()` では，定義を行う領域の各アドレスを *def* に追加していく（2）．ここで，アドレスは 32bit 単位とした．`__slice_inst()` でも，最初に *use* と *def* の更新を行う（3）,（4）．これは，レジスタを介した Def-Use も考慮する必要があるためである．ただし，スタックレジスタとフレームポインタ（ESP と EBP）の使用と定義は除外した．これらのレジスタも含めると，スタックの操作を行うすべての命令がスライスに含まれてしまうためである．次に，現在の命令だけを含む集合と，

[2]roBDD とは，既約な順序付き二分決定図であり，集合の表現に適した特徴を持つ．

```
Let slice be the set of slices.
Let use be the set of slices for addresses used.
Let def be the set of addresses defined.
__slice_use() {
    foreach address in the access area
        Add slice[address] to use.  (1)
    end
}
__slice_def() {
    foreach address in the access area
        Add address to def.  (2)
    end
}
__slice_inst() {
    foreach register used
        Add slice[register] to use.  (3)
    end
    foreach register defined
        Add register to def.  (4)
    end
    Initialize the set new_slice to include
    only the current instruction.  (5)
    foreach used_slice in use
        new_slice = new_slice ∪ used_slice  (6)
    end
    foreach address in def
        slice[address] = new_slice  (7)
    end
    Reset use & def.  (8)
}
```

図 4.1 スライスの計算

use に含まれるすべてのスライスの和集合を取る（5）,（6）．これが，現在の命令が依存する新しいスライスとなる．そして，*def* に含まれるすべてのアドレスのスライスを，計算したスライスに更新する（7）．また，次の命令に備えて *use* と *def* をリセットしておく（8）.

コマンド　スライスの計算には，非常に大きな時間がかかる．またスライスの計算を行う実行区間が長くなると，スライスのサイズも大きくなる傾向がある．そのためスライシングの機能は，デバッガのユーザが必要に応じ有効/無効を切り替え，最小限の実行区間にだけ適用することが望ましい．そこでデバッガに，次のコマンドを用意した．

```
slice start [スレッド]
slice stop
slice get 対象変数やメモリ領域
```

start コマンドと stop コマンドは，それぞれスライシングを開始・終了するコマンドである．start コマンドは，引数に指定されたスレッドに対し，上述のイベントハンドラを有効にする．引数を省略した場合には，すべてのスレッドが対象となる．また stop コマンドは，イベントハンドラを無効にする．そして get コマンドは，図 4.1 の *slice* にアクセスし，現在の停止位置までに計算されたスライスを取得するコマンドである．最後に取得したスライスに含まれる行は，デバッギの表示するソースコード上でマークが付与される．

マルチスレッド　最後に，マルチスレッドプログラムに関連して行った拡張について述べる．マルチスレッドプログラムには，複数のスレッドが同じ処理を行うものも多い（サーバなど）．そのようなプログラムに対してスライシングを行うと，スライスに含まれる命令がどのスレッドによるものなのか，判別がつかなくなってしまう．そのため本研究では，図 4.1（5）の処理を変更し，カレントスレッドの識別子と現在の命令をエンコードしたものを，*new_slice* として初期化するようにした．これによりスライスに含まれる命令は，スレッドごとに区別されるようになる．またデバッガにも，次のコマンドを用意した．

```
slice thread スレッド
```

これは，get コマンドで取得するスライスを，特定のスレッド識別子を持つものに限定するためのコマンドである．

4.2　可逆実行

本節では，可逆実行の概要と，本研究での実装について述べる．

4.2.1 概要

可逆実行とは，通常のプログラムの実行方向とは，逆向きの実行（逆実行）を可能にする機能である．可逆実行の実現方式には，次の 2 種類の方式が存在する．

- **ロギング方式** ロギング方式 [1, 2, 14, 16, 20, 21, 44] は，デバッギの実行中に，変更された状態の差分を保存していく方式である．逆実行は，この差分情報を基に，過去の状態を復元していくことにより実現される．例えば，デバッギが，メモリの上書きを行う命令を実行する場合を考える．この場合には，通常方向の実行時に，メモリのアドレスと変更前の内容を保存しておく．そして逆実行時には，メモリの内容を変更前のものに復元する．

- **再実行方式** 再実行方式 [10, 73, 41] は，デバッギを再実行し，（現在の停止位置より前の）適切な位置で停止させる方式である．そのため，実際に逆方向に実行しているわけではなく，疑似的な方式である．デバッギのコードには，ループや関数呼出しなどにより，何度も繰り返し実行されるコードが多く含まれる．そのようなコードに対し，逆実行を行う際には，現在の時点から見て，最後の実行の時点で停止させる必要がある．

4.2.2 方針

ロギング方式と再実行方式には，それぞれ一長一短がある．ロギング方式では，デバッギの通常方向の実行時に，実行速度とメモリの両面で，状態の保存に伴う比較的大きなオーバーヘッドが発生する．しかし，逆実行は比較的高速である．これに対し，再実行方式では，通常方向の実行時に発生するオーバーヘッドが比較的小さい．しかし，逆実行にかかる時間は，再実行をする区間の長さに依存する．そのため，逆実行をする区間が短くても，長い時間が必要となる場合もある[3]．そこで本研究では，両方の方式を実装し，どちらの方式を利用するか，デバッガのユーザが選択できるようにした．また実装を簡略化するため，どちらの方式でも，既にデバッギの実行が記録されており，その再生中に利用されることを前提とした．

4.2.3 ロギング方式の実装

制限 マルチスレッドプログラムでは，実行状態として，スレッド機構の状態も考慮する必要がある．しかし，スレッド機構の状態は非常に複雑に構成されているため，ロギング

[3] この問題を改善するために，定期的にチェックポイントを保存する手法も提案されている [10].

表 4.2 内部状態の差分を保存するためのイベントハンドラ

イベント	ハンドラ
LINE, PROC	`void __log_top(dword inst);`
STORE	`void __log_store(dword inst, dword addr, dword size);`

方式を適用することは簡単ではない．そのため現在の実装では，シングルスレッドプログラムだけを対象とすることにした．マルチスレッドプログラムへの対応は，今後の課題である．

イベントハンドラ　デバッギの実行状態には，内部状態と外部状態がある．内部状態とは，レジスタやメモリなど，プログラム内部の状態のことである．これに対し，外部状態とは，ファイルシステムなど，プログラム外部の状態のことである．

本研究では，内部状態の差分を保存するために，表 4.2 に示した 2 つのハンドラを利用する[4]．`__log_top()` は，ソースコードにおける行，または関数の先頭で呼び出されるハンドラである．また`__log_store()` は，メモリの上書きが行われる際に呼び出されるハンドラである．命令が行，または関数の先頭であり，さらにメモリの上書きも行う場合には，まず`__log_top()` が呼び出され，次に`__log_store()` が呼び出される．

図 4.2 に，これらのハンドラの処理を疑似コードで表したものを示す．図 4.2 において，def は既に変更前の内容を保存したメモリ領域に関する情報を保持する．`__log_top()` では，まず現在の命令が，後述する逆実行単位の先頭であるか検査を行う（図 4.2（1））．もしそうであるならば，デバッギのレジスタの内容をロギング用のバッファに保存する（2）．また def もリセットする（3）．`__log_store()` では，まず上書きを行う領域が，既に保存されていないか検査を行う（4）．もしまだであれば，領域の内容をロギング用のバッファに保存する（5）．そして領域を def に追加する（6）．

外部状態は，主にシステムコールの実行結果に影響を及ぼすものである．しかし本研究のデバッガでは，実行の記録・再生を行うため，システムコールの実行結果は，外部状態に関わらず常に一定となる．そのため，外部状態に関しては保存を行う必要がない．代わりに，実行の再生に用いるデータ（SIC（Software Instruction Counter）と記録ファイルの再生位置）の保存を行う．これは，図 4.2 の（2）の処理に追加した．

コマンド　状態の保存には，比較的大きなオーバーヘッドが伴う．そのため，スライシングの場合と同様に，必要に応じて有効/無効を切り替えられることが望ましい．そこでデバッガに，次のコマンドを用意した．

[4] メモリの割当て（mmap() など）を追跡するためのハンドラなども存在するが，ここでは省略する．

```
Let def be the information of saved memory area.
__log_top() {
    if the current instruction is a beginning
    of the reverse execution unit then (1)
        Save the registers to the logging buffer.  (2)
        Reset def.  (3)
    end
}
__log_store() {
    if not the access area ∈ def then (4)
        Save the access area to the logging buffer.  (5)
        Add the area to def.  (6)
    end
}
```

図 4.2 内部状態の保存

```
log enable 単位
log disable
```

enable コマンドは，上述したイベントハンドラを有効にし，状態の保存を開始するコマンドである．enable コマンドには，引数として逆実行の単位を指定する．ここには，ソースコードにおける行や関数が指定可能である．disable コマンドは，イベントハンドラを無効にするコマンドである．

また逆実行の単位を，最後に取得したスライスに含まれる行にすることも可能である．その場合には，次のコマンドを利用する．

```
log slice
```

slice コマンドでは，まずデバッギを再起動する．次に，スライシングを開始した時点まで実行を継続し，そこで状態の保存を開始する．そして，対象スライスを取得した時点まで実行を継続する．これらの実行の制御には，次節で述べる SIC と PC のペアを利用している．

逆実行は，次のコマンドで行う．

```
log bstep ステップ数
log bcont
```

bstep コマンドは，逆方向にステップ実行を行うコマンドである．bstep コマンドは，ロギングを行ったバッファを後ろから順番にたどっていき，過去の状態を復元していく．ま

表 4.3 SIC の比較を行うイベントハンドラ

イベント	ハンドラ
SIC	`void __sic_cmp(dword inst, qword sic, int jmp);`

た bcont コマンドは，ブレークポイントにヒットするまで，逆方向に継続実行を行うコマンドである．bcont コマンドは，bstep コマンドと同様の処理を繰り返しながら，ブレークポイントの検査を行う．以上の処理の大半は，デバッガではなく，デバッギのプロセス空間で行われる．

4.2.4 再実行方式の実装

一般的に，再実行方式の実装には，次の 2 つの課題がある．

- 決定的なプログラム実行を実現する機構の実現
- プログラムを適切な位置で停止させる機構の実現

前者の機構は，既に DbgStar に組み込まれている．そこで本研究では，後者の機構を実装した．実装は，停止条件の指定方法の違いにより，SIC と PC のペア，フレーム識別子，スライスの 3 種類に分類できる．いずれの停止条件でも，マルチスレッドプログラムの逆実行を行うことが可能である．

4.2.4.1 SIC と PC のペア

3.3.3 節で述べた SIC と PC のペアにより，停止条件を指定する．重要なタイミング（スライシングの開始など）で SIC と PC の値を保存しておけば，いつでもそこに巻戻ることが可能となる．

イベントハンドラ　使用するハンドラは，表 4.3 に挙げた 1 種類だけである．`__sic_cmp()` は，SIC がインクリメントされるたびに呼び出されるハンドラである．`__sic_cmp()` では，現在の SIC と停止条件の SIC を比較し，一致していたらデバッギを停止させる．

コマンド　デバッガには，次のコマンドを用意した．

```
sic get
sic cont SIC [PC]
sic bcont SIC [PC]
```

表 4.4 スタックフレームを追跡するイベントハンドラ

イベント	ハンドラ
CALL	`void __fid_call(dword caller, dword callee, dword ret);`
RET	`void __fid_ret(dword callee, dword caller);`

get コマンドは，現在の SIC と PC の値を出力するコマンドである．また cont コマンドは，指定された SIC と PC の値になるまで，デバッガの実行を継続するコマンドである．cont コマンドは，まず上述したハンドラを利用し，指定された SIC になるまで実行を継続する．そしてブレークポイントを利用し，指定された PC まで実行を継続する．bcont コマンドは，まずデバッギを再起動してから，cont コマンドと同様の処理を行うコマンドである．

4.2.4.2 フレーム識別子

プログラムは，関数を呼び出すと，スタックにフレームと呼ばれる領域を割り当てる．また関数から復帰すると，フレームを解放する．そのため，過去のスタックフレームを調査することにより，関数の呼出し元をたどっていくことが可能である．本研究では，丸山ら [73] と同様の手法を利用し，特定のスタックフレームに対応する関数の先頭で，デバッギを停止させる機能を実装した．

イベントハンドラ　イベントハンドラは，スタックフレームの割当てと解放を追跡する．これは，主に表 4.4 に挙げた 2 つのハンドラで行う．`__fid_call()` と `__fid_ret` は，それぞれ関数の呼出しと復帰によって呼び出されるハンドラである．これらのハンドラでは，フレーム識別子と，独自の呼出しスタックを管理する．`__fid_call()` では，フレーム識別子をインクリメントし，その値を呼出しスタックにプッシュする．`__fid_ret()` では，呼出しスタックをポップする（値は捨てる）．また`__fid_call()` では，現在のフレーム識別子と停止条件のフレーム識別子を比較し，一致していたらデバッギを停止させる．

コマンド　次章で例を示すように，現在の関数の先頭や，呼出し元の関数の先頭に戻る機能は，デバッギの実行を調査していく上で非常に便利な機能である．また上述のハンドラは，オーバーヘッドも比較的小さい．そのため，本研究で実装したデバッガでは，上述のハンドラを常に有効にしている．

デバッガには，次のコマンドを用意した．

表 4.5 スライスに含まれる行の実行回数を計測するイベントハンドラ

イベント	ハンドラ
INST	`void __rev_inst(dword inst, dword load, dword store);` `int __rev_inst_check(dword addr)`

```
fid dump
fid cont フレーム識別子
fid bcont フレーム識別子
```

dump コマンドは，ハンドラが管理しているスタックをダンプし，フレーム識別子を表示させるコマンドである．cont コマンドは，指定されたフレーム識別子になるまで，デバッギの実行を継続するコマンドである．また bcont コマンドは，まずデバッギを再起動してから，cont コマンドと同様の処理を行うコマンドである．

4.2.4.3 スライス

スライスに含まれる行（最後に取得したもの）に沿って，デバッギの実行を制御する．

イベントハンドラ イベントハンドラは，スライスに含まれる行の実行回数を計測（プロファイリング）する．これは，主に表 4.5 に示した__rev_inst() というハンドラで行う．デバッガは，__rev_inst() ハンドラの登録時のオプションに，__rev_inst_check() という関数を渡す．__rev_inst_check() は，引数に渡されたアドレスを検査し，スライスに含まれる行の先頭命令のアドレスである場合に真を返す．仮想マシンは，__rev_inst() の instrumentation 時に，__rev_inst_check() を呼び出す．そして，戻り値が真である場合にだけ instrumentation を行う．そのため，__rev_inst() は，スライスに含まれる行の実行時に呼び出されるハンドラとなる．

__rev_inst() は，スライスに含まれる行に対して，次の実行回数を計測する．

- すべての行の実行回数の総計
- 個別の行の実行回数
- ブレークポイントにヒットした行の実行回数

これらの実行回数は，スレッドのタイムインターバル（実行権が与えられてから，失われるまで）ごとに計測される．また__rev_inst() では，これらの実行回数と停止条件を比較し，一致していたらデバッギを停止させる．

コマンド　デバッガのユーザは，まずスライスに含まれる行のプロファイリングを行う．これは，次のコマンドで行う．

```
slice profile
slice dump_profile
```

profile コマンドでは，まずデバッギを再起動する．次に，スライシングを開始した時点まで実行を継続し，そこで上述のハンドラを有効にする．そして，対象スライスを取得した時点まで実行を継続する．また dump_profile コマンドは，スレッドのタイムインターバルごとに，計測結果の概要を表示するコマンドである．

スライスに沿った実行制御は，次のコマンドで行う．

```
slice step ステップ数
slice cont
slice bstep ステップ数
slice bcont
slice first タイムインターバル
slice last タイムインターバル
```

step コマンドと cont コマンドは，スライスに沿ってステップ実行と継続実行を行うコマンドである．また bstep コマンドと bcont コマンドは，それらを逆方向に行うコマンドである．first コマンドと last コマンドは，指定されたタイムインターバルにおいて，スライスに含まれる行が最初，もしくは最後に実行された時点で停止するコマンドである．

これらのコマンドでは，profile コマンドで計測した情報と，現在の停止位置までに計測した情報に基づいて，停止条件を計算する．そして，必要に応じデバッギを再起動し，停止条件を満たすまでデバッギを実行する．例えば bcont コマンドでは，まず最後にブレークポイントにヒットしたタイムインターバルと，そのタイムインターバルでブレークポイントにヒットした回数を計算する．次に，デバッギを再起動し，そのタイムインターバルまで実行する．そして__rev_inst() を利用し，ブレークポイントのヒット回数を計測し，先程計算した回数に達したらデバッギを停止する．

第5章　実 験

2 章で述べた 3 つの要件に基づいて，DbgStar の評価・考察を行う．本章では，まず DbgStar のオーバーヘッドについての評価結果を示す．次に，前章で紹介したデバッガを用いたデバッグシナリオを示す．そして，デバッグシナリオを通し，DbgStar の親和性と柔軟性について考察を行う．

5.1　オーバーヘッドの評価

本研究では，DbgStar の仮想マシンによるオーバーヘッドと，実行の記録・再生に伴うオーバーヘッドの評価を行った．評価を行ったシステムの構成は，CPU Pentium4 2.4GHz, Memory 1.0GB，Linux Kernel 2.4.35，GCC 3.3.5，GLIBC 2.3.2 である．

5.1.1　仮想マシンによるオーバーヘッド

他の手法との比較　DbgStar のオーバーヘッドを，LVM（2.2.5 節）と Intel ATOM 1.0 Beta（2.2.6 節）のオーバーヘッドと比較した．ここで ATOM に関しては，Linux Kernel 2.4.35 上で動作しなかったため，Linux Kernel 2.6.8 上で計測した．評価では，ベンチマークとして Stanford Integer Benchmark Suite を利用した．これは，LVM には互換性の問題があり，SPEC CPU2000 を動かせなかったためである．また 2.2.5 節で述べたように，Stanford Integer Benchmark Suite を動かすためにも，ベンチマークのコードを修正する必要があった．修正したコードが，ベンチマーク全体の実行時間に与える影響は，無視できる範囲のものである．表 5.1 に，評価結果を示す．表中の数字は，各ベンチマークを CPU 上で実行した場合の実行時間と，それぞれの環境下で実行した場合の実行時間の比率である．Base は監視を全く行わなかった場合，ST（Store）はすべてのメモリの書込みを監視した場合，LD-ST（Load-Store）はすべてのメモリの読書きを監視した場合を表す．平均は，幾何平均を表す．また図 5.1 に，特に ATOM と DbgStar の数値をグラフで比較したものを示す．

表 5.1 に示したように，LVM 上でベンチマークを実行すると，監視を全く行わなかった場合でも平均 85.7 倍の実行時間の増加が見られた．またメモリの書込みと読書きを監視した場

表 5.1 他の手法との比較

	LVM			ATOM		DbgStar		
ベンチマーク	Base	ST	LD-ST	ST	LD-ST	Base	ST	LD-ST
bubble	181.9	506.7	1777.3	12.4	57.3	8.1	22.0	82.9
hanoi	118.3	347.5	1026.8	19.8	40.0	11.7	41.6	81.0
matmul	77.5	210.1	873.1	6.7	36.8	5.4	11.2	49.5
perm	108.1	356.8	1010.8	9.7	29.4	9.3	21.9	56.0
qsort	117.8	357.0	1345.1	9.1	45.0	4.7	15.4	63.7
queen	103.7	274.2	1010.7	7.3	33.6	6.2	14.7	50.9
sieve	15.4	57.8	169.1	1.8	4.8	2.1	3.6	8.8
平均	85.7	257.3	867.6	7.9	29.1	6.0	15.2	47.4

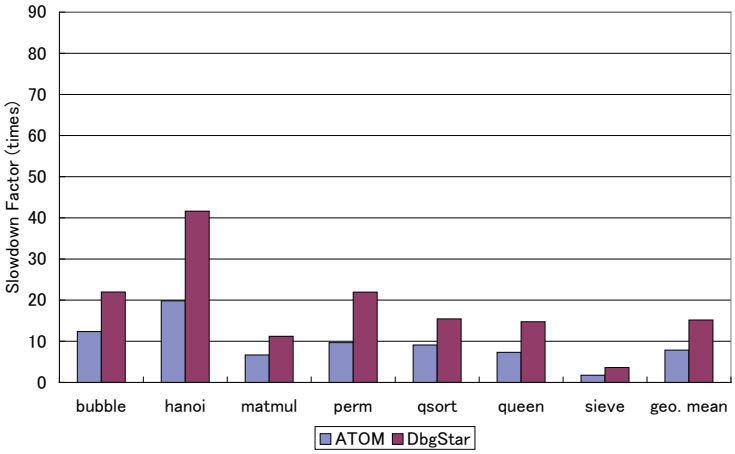
（CPU 上で直接実行した場合の実行時間との比率）

合，それぞれ平均 257.3 倍と 867.6 倍の実行時間の増加が見られた．これに対して，ATOM では，監視を全く行わない場合には，オーバーヘッドが発生しない（instrumentation を行う必要がないため）．またメモリの書込みと読書きを監視した場合，それぞれ平均 7.9 倍と 29.1 倍の実行時間の増加が見られた．このことから，逐次解釈実行の手法に基づいた LVM と，静的な instrumentation の手法に基づいた ATOM とでは，ATOM のオーバーヘッドの方がはるかに小さいことがわかる．

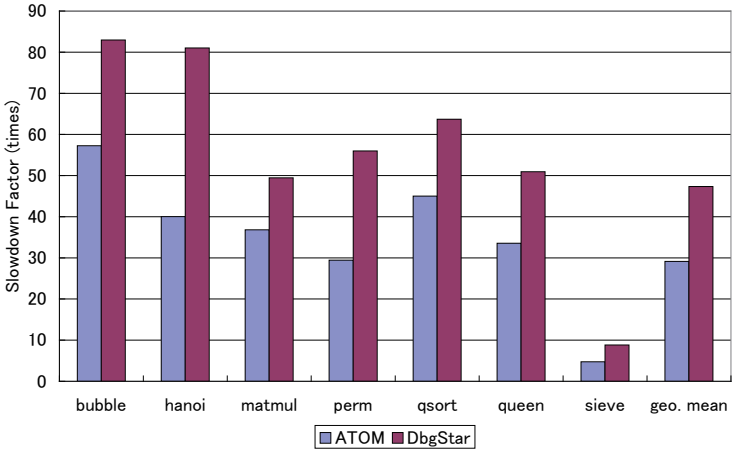
次に，DbgStar のオーバーヘッドを調べると，監視を全く行わなかった場合で平均 6.0 倍の実行時間の増加が見られた．これは，ATOM と比較すると 6.0 倍の実行時間の増加であるが，LVM と比較すると 14.3 倍の実行時間の短縮である．またメモリの書込みと読書きを監視した場合，それぞれ平均 15.2 倍と 47.4 倍の実行時間の増加が見られた．これは，ATOM と比較すると 1.9 倍と 1.6 倍の実行時間の増加であるが，LVM と比較すると 16.9 倍と 18.3 倍の実行時間の短縮である．このことから DbgStar は，ATOM には及ばないものの，LVM よりはるかに小さいオーバーヘッドを実現していることがわかる．

他の仮想マシンとの比較　DbgStar のオーバヘッドを，Valgrind[45, 46] のオーバーヘッドと比較した．Valgrind は，DbgStar と同様に，仮想マシンを利用して動的なコード変換を行うフレームワークである．ただし，DbgStar がデバッガの構築に特化しているのに対し，Valgrind は Shadow Value Tool[46] の構築に特化している[1]．Shadow Value とは，個々のレジスタやメモリアドレスの状態を表す値である．Shadow Value Tool は，プログラム実行を監視し，Shadow Value を更新・追跡していくことにより，何らかの解析を行うツールのことである．Valgrind を利用して構築された Shadow Value Tool の例として，CPU

[1]例えば，Valgrind では，ブレークポイントやステップ実行など，デバッガに必須の機能が提供されていない．そのため，デバッガの構築には利用することができない．



(a) すべてのメモリの書込みを監視した場合



(b) すべてのメモリの読書きを監視した場合

図 5.1 ATOM と DbgStar の比較

のキャッシュシミュレーションを行う Cachegrind や，メモリエラーを検査する Memcheck などが存在する [64].

図 5.2 と図 5.3 に，Valgrind 2.4.1 と DbgStar のオーバーヘッドの評価結果を示す．図 5.2 は，SPEC CPU2000[56] の 8 つのプログラム（シングルスレッド）の評価結果である．また図 5.3 は，SPLASH-2[69] の 4 つのプログラム（マルチスレッド）の評価結果である．ここで，SPEC CPU2000 は，トレーニングサイズの入力データを用いて実行した．また，SPLASH-2 に指定したオプションの一覧を，表 5.2 に示す．これらのオプションでは，各ベンチマークが解く問題のサイズ（`-n16777216`，`-batch -room`，`inputs/balls4.env`，`inputs/head`）や，8 個のスレッドを使って問題を解くこと（(`-p`) `8`）などを指定している．図 5.2 と図 5.3 の Y 軸は，各ベンチマークを CPU 上で実行した場合の実行時間と，それぞれの環境下で実行した場合の実行時間の比率である．

まず，SPEC CPU2000 の評価結果（図 5.2）について考える．監視を全く行わなかった場合，Valgrind で平均 4.6 倍（幾何平均，以下同様），DbgStar で平均 4.2 倍の実行時間の増加が見られた．そのため，監視を全く行わなかった場合には，実行速度にあまり差はないことがわかる．これに対し，メモリの読込みと書込みを監視した場合，Valgrind でそれぞれ平均 12.3 倍と 8.0 倍，DbgStar でそれぞれ平均 38.3 倍と 17.3 倍の実行時間の増加が見られた．そのため，実行の監視を行った場合には，DbgStar の速度が，Valgrind と比較して相対的に悪化することがわかる．

これは，Valgrind の仮想マシンが行う最適化に関係していると考えられる．Valgrind では，コード変換時に，対象プログラムのコードと監視コードをいったん中間表現に変換する．そして，中間表現をネイティブコードに変換する際に，様々な最適化を行う．これに対して，DbgStar では，変換したコードに対しあまり最適化を行っていない．そのため，実行時間にこのような差が出たものと考えられる．DbgStar における変換コードの最適化は，今後の課題である．

次に，SPLASH-2 の評価結果（図 5.3）について考える．SPLASH-2 の評価結果には，ベンチマークごとに非常に大きなばらつきがある．例えば，volrend で監視を全く行わなかった場合，Valgrind で 12.6 倍の実行時間の増加が見られた．これに対し，DbgStar では，ほとんどオーバーヘッドが発生していない．これは，volrend がビジーウェイトのために消費する時間に関係していると考えられる．ビジーウェイトとは，特定の条件が満たされるまで，その場でループして待つウェイトのことである．DbgStar では，スレッドのスケジューリングが非常に短い間隔で行われる．そのため，ビジーウェイトのために消費された時間が，CPU 上で実行した場合と比較して短かったものと推測される．そして，ビジーウェイトの部分の高速化が，DbgStar のその他の部分による実行速度の低下と丁度相殺しあったものと考えられる．

表 5.2 SPLASH-2 に指定したオプション

ベンチマーク	オプション
radix	`-p 8 -n16777216 -t`
radiosity	`-p 8 -batch -room`
raytrace	`-p8 inputs/balls4.env`
volrend	`8 inputs/head`

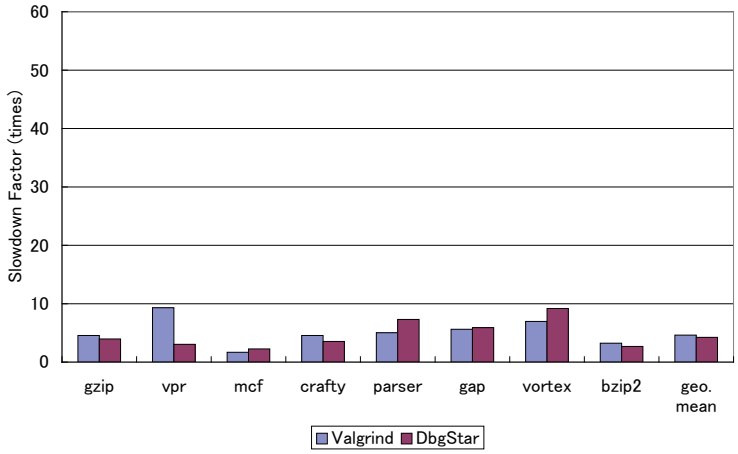
このことから，マルチスレッドプログラムに関しては，スケジューリングのポリシーも実行時間に大きな影響を与えることがわかる．図 5.3 に示したように，今回の実験では，DbgStar のスケジューリングポリシーがたまたま良い結果を示した．しかし，最適なスケジューリングポリシーの研究は，今後の課題である．

まとめ　本節では，逐次解釈実行の手法に基づいた LVM のオーバーヘッドと比較すると，DbgStar のオーバーヘッドの方がはるかに小さいことを示した．しかし，静的な instrumentation の手法に基づいた ATOM のオーバーヘッドと比較すると，DbgStar のオーバーヘッドの方が大きい．これは，ATOM が実行前に行っている instrumentation を，DbgStar では仮想マシンを利用して実行時に行っているため，ある程度は仕方のないことである．DbgStar は，このオーバーヘッドと引き替えに，後述する柔軟性を実現する．また，Valgrind との比較で示したように，DbgStar のオーバーヘッドには改善の余地がある．そのため，ATOM との差は，まだ縮めることが可能であると期待される．

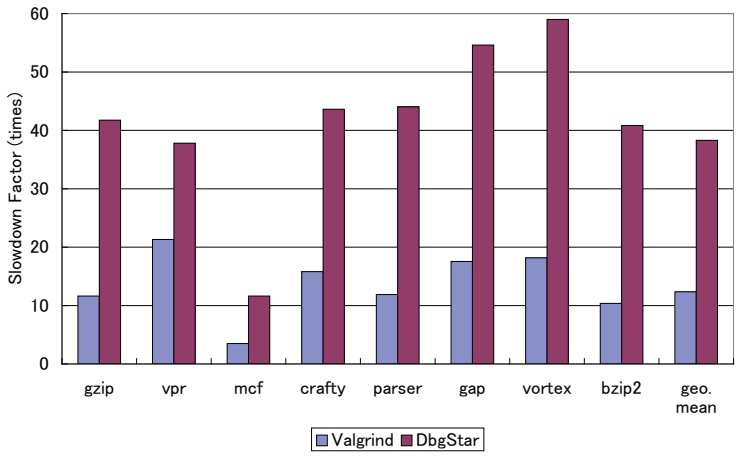
5.1.2　実行の記録・再生に伴うオーバーヘッド

実行の記録･再生に伴うオーバーヘッドの評価結果を表 5.3 に示す．評価には，SPLASH-2 の 4 つのプログラムを使用した．表 5.3 において，Record Size は，実行の記録に使用された容量を KB 単位で表す．Syscall，Signal，Schedule は，それぞれシステムコール，シグナル，スレッド機構の振舞いの記録に使用された容量である．また Total は，それらの総計である．表 5.3 に示したように，radix と radiosity では，スケジューリングの記録にほとんどの容量が使われている．これに対し，raytrace と volrend では，システムコールの実行結果の記録にも比較的大きな容量が使われている．これらは，raytrace と volrend の入力ファイルの記録によるものである．

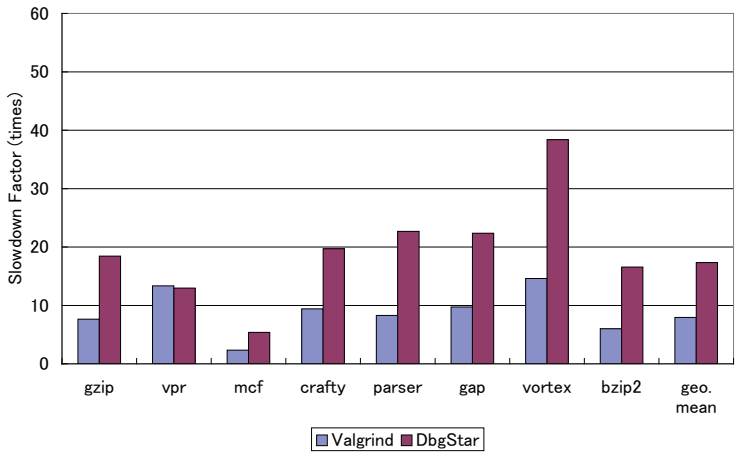
また Execution Time は，実行の記録・再生にかかった実行時間と，監視をまったく行わなかった場合の実行時間（図 5.3（a））の比率である．表 5.3 に示したように，記録実行時には，ほとんど実行時間の増加が見られなかった（平均 1.01 倍）．また再生実行時には，実行時間の短縮が見られた．この短縮は，スケジューリングなどの高速化によるもの



（a）監視を全く行わなかった場合

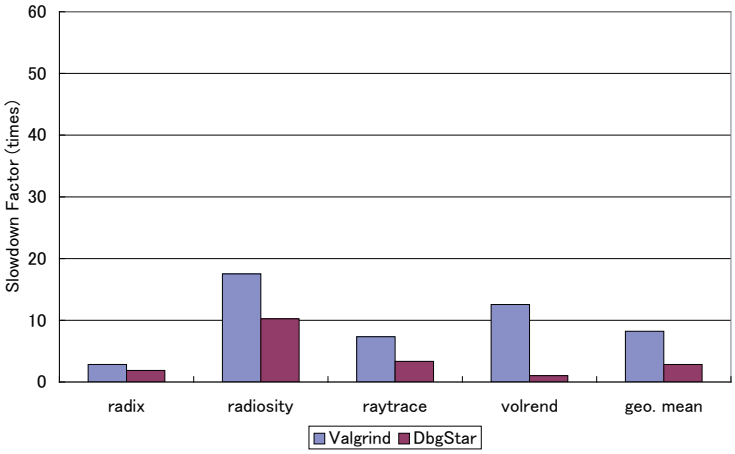


（b）すべてのメモリの読込みを監視した場合

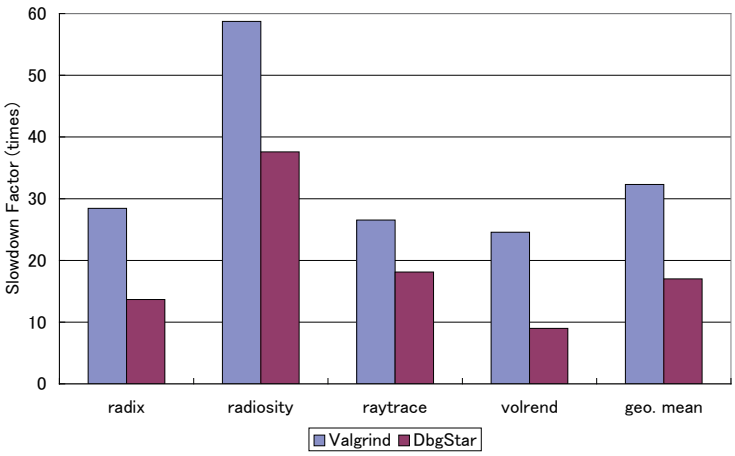


（c）すべてのメモリの書込みを監視した場合

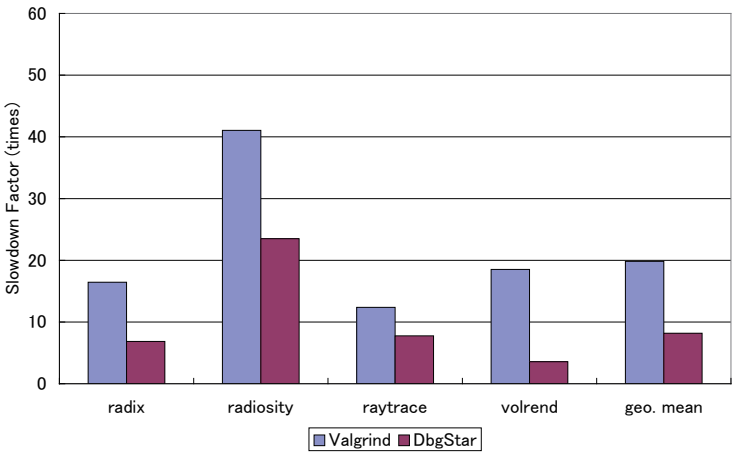
図 5.2 Valgrind と DbgStar の比較（SPEC CPU2000）



（a）監視を全く行わなかった場合



（b）すべてのメモリの読込みを監視した場合



（c）すべてのメモリの書込みを監視した場合

図 5.3 Valgrind と DbgStar の比較（SPLASH-2）

表 5.3 実行の記録・再生に伴うオーバーヘッド

	Record Size (KB)				Execution Time (relative)	
ベンチマーク	Syscall	Signal	Schedule	Total	Record	Replay
radix	2.0	0	1.2×10^2	1.3×10^2	1.001	0.949
radiosity	5.3	0	1.2×10^4	1.2×10^4	1.005	0.877
raytrace	1.2×10^3	0	2.7×10^3	3.9×10^3	1.002	0.895
volrend	2.2×10^4	0	2.6×10^2	2.2×10^4	1.033	0.891
平均	-	-	-	-	1.010	0.903

と考えられる．仮想マシンのスケジューラでは，次に実行するスレッドを選択する処理を行う．しかし再生実行時には，次に実行するスレッドが最初から決定されている．そのため，処理の大半をスキップすることが可能である．

5.2 デバッグシナリオと親和性・柔軟性の考察

本節では，前章で紹介したデバッガを用いたデバッグシナリオを示す．そして，デバッグシナリオを通し，DbgStar の親和性と柔軟性について考察を行う．シナリオでは，ProFTPD, GNU Awk，Apache HTTP Server の 3 つのプログラムに含まれていた実際の不具合を利用する．なお，ここで示すシナリオは，それぞれの不具合がどのようにして起こったかを，デバッガを用いて解析するシナリオである．実際にそれぞれの不具合をどのようにして修正するかは，デバッガのスコープ外であることに注意されたい．

5.2.1 ProFTPD

ProFTPD[62] は，広く利用されている FTP サーバである．ProFTPD は，モジュール化された設計を採用しており，様々な機能拡張モジュールを提供している．

不具合の概要　ProFTPD 1.2.10 の `mod_radius` モジュールには，不正なメモリ参照により強制終了する不具合が存在する [61]．`mod_radius` は，ProFTPD から RADIUS プロトコルを利用するためのモジュールである．RADIUS は，認証とアカウンティングのためのプロトコルである．`mod_radius` モジュールの不具合は，RADIUS 認証を利用してログインを行う際に，非常に長いパスワードを入力すると発生する．

デバッグシナリオ　デバッガを利用するために，ProFTPD の構築時に，`configure` スクリプトに対していくつかのオプションを指定した．まず指定したオプションは，デバッグ

情報を生成するためのものと，3.2.1.1 節で述べたライブラリをリンクするためのものである．また ProFTPD は，セッションごとに `fork()` を行い，子プロセスを生成するサーバである．しかし 3.3.2.1 節で述べたように，DbgStar では `fork()` の実行を禁止している．そのため，子プロセスを生成しないようにする開発者用のオプションも指定した．ProFTPD のソースコードに対しては，特に修正を行う必要はなかった．

デバッグは，次のステップで行った．

(1) 不具合の記録と再現 まず，不具合の起きた実行を記録するために，RADIUS 認証を利用して ProFTPD にログインを行った．そして，非常に長いパスワード（512 文字）を入力したところ，ProFTPD はセグメンテーションフォルトによって強制終了された．そこで，記録した実行をデバッガ上で再現した．ここで，記録ファイルのサイズは 164.7KB であった．これは，すべてシステムコールの実行結果の記録によるものである．また実行の記録・再生にかかった時間は，それぞれ 1.4 秒と 1.2 秒であった．この時間は，ユーザ時間とシステム時間の合計であり，パスワードなどの入力待ちの時間は除外されたものである．

(2) 状況の確認 デバッギは，不正な参照を行った時点で停止している．不正な参照は，`MD5Update()` 内の `MD5_memcpy()` の呼出しによって発生していた（図 5.4）．この呼出しでは，`&input[i]` から`&context->buffer[index]` へ，メモリの内容がコピーされる．そこで，`input` の値を調査したところ，値は 0x02 であった．これは，明らかに不正なメモリアドレスである．

さらにスタックフレームを調査すると，`MD5Update()` は，`radius_add_passwd()` から呼び出されていることがわかった（図 5.5：1380 行目）．`MD5Update()` の呼出しの 2 番目の引数（`digest`）が，図 5.4 の `input` に対応する．`digest` の値を確認したところ，値はやはり 0x02 であった．

(3) スライシング ここで，スライシングを行ってみることにした．そのために，まず `radius_add_passwd()` の先頭まで逆実行した．この逆実行には，4.2.4.2 節で紹介したフレーム識別子を利用した．逆実行には，1.4 秒かかった．次にスライシングを有効にし，デバッギの実行を再開した．そして，不正な参照で停止するまで，スライスの計算を行った．スライスの計算には，0.03 秒かかった．

(4) スライスの調査 まず，`digest` の値に対するスライスを調査した．このスライスには，ネイティブコードで 6 命令，ソースコードに換算して 4 行が含まれていた．ソースコードの 4 行を調査したところ，`digest` の値に直接関係がありそうなのは，図 5.5 において S と印をつけた 2 行であった．しかし，`radius_add_passwd()` の呼出し元

```
        static void MD5Update(MD5_CTX *context,
                              unsigned char *input,
                              unsigned int inputLen)
        {
            ...
1073:       MD5_memcpy(&context->buffer[index],
                       &input[i], inputLen-i);
        }
```

図 5.4 MD5Update() 関数（textttmod_radius.c）

を調べると，引数 type には RADIUS_PASSWORD が指定されていた．そのため 1371 行目ではなく，1368 行目が実行されるはずである．

そこで，type の値に対するスライスを調査した．このスライスには，ネイティブコードで 30 命令，ソースコードに換算して 4 行が含まれていた．ソースコードの 4 行は，図 5.5 において*と印をつけた行である．しかし，この 4 行は，一見 type の値に関係がなさそうである．ここで，pwlen の値を調査したところ，値は 512 であった．次に，pwhash のサイズを確認したところ，サイズは 272 バイト（RADIUS_PASSWORD_LEN は 16 バイト）であった．このことから，1362 行目の memcpy() で，pwhash の境界を超えてメモリのコピーが行われたことがわかる．その際に，type の値も破壊されてしまったと推測される．実際に，pwhash と type のアドレスを調査すると，この推測が正しかったことが確認できた．

5.2.2 GNU Awk

AWK[4] は，A. Aho，P. Weinberger，B. Kernighan によって開発されたテキスト処理スクリプト言語である．GNU Awk[25] は，GNU Project によって実装された AWK の処理系の 1 つである．

不具合の概要 GNU Awk 3.1.5 には，不正なメモリ解放の不具合が存在する [40]．例えば，次のコマンドを入力したとする．

```
$ gawk 'n+=length($1) END print n' < small.txt
```

これは，入力（small.txt）の各行の最初の単語の長さ（length($1)）を n に加算していき，最後にその合計を出力するスクリプトである．ここで small.txt は，次の内容のファイルである．

```
        static void radius_add_passwd(
            radius_packet_t *packet, unsigned char type,
            const char *passwd, char *secret)
        {
            ...
1343:       char pwhash[256 + RADIUS_PASSWD_LEN];
1344: *     size_t pwlen = strlen(passwd);
            ...
1354: *     pwlen += (RADIUS_PASSWD_LEN - 1);
            ...
1357: *     pwlen &= ~(RADIUS_PASSWD_LEN - 1);
            ...
1362: *     memcpy(pwhash, passwd, pwlen);
            ...
1365: S     attrib = radius_get_attrib(packet, RADIUS_PASSWORD);
1367:       if (type == RADIUS_PASSWORD)
1368:           digest = packet->digest;
1369:
1370:       else
1371: S         digest = attrib->data;
            ...
1380:       MD5Update(&ctx, digest, RADIUS_VECTOR_LEN);
        }
```

図 5.5 radius_add_passwd() 関数（mod_radius.c）

```
1:
2:  hello
3:  world
```

最初の行は，空行である．すると筆者らの環境では，次のメッセージが表示され，GNU Awk が強制終了された．

```
*** glibc detected *** double free or corruption (fasttop):
0x080993a8 ***
アボートしました
```

デバッグシナリオ　デバッガを利用するために，GNU Awk の構築時に，configure スクリプトに対していくつかのオプションを指定した．指定したオプションは，デバッグ情報を生成するためのものと，3.2.1.1 節で述べたライブラリをリンクするためのものである．GNU Awk のソースコードに対しては，特に修正を行う必要はなかった．また，上述の small.txt では，非常に短い実行に対するデバッグシナリオしか示すことができない．そのため，ここでは，マルコフモデルに基づいてランダム生成したテキストファイル（large.txt）を入力として用いた．large.txt のファイルサイズは，4.0MB である．large.txt の各行は単一の単語からなり，末尾の数行手前に空行が存在する．

デバッグは，次のステップで行った．

（1）不具合の記録と再現　まず，上記のコマンドと large.txt を用い，不具合の起きた実行を記録した．そして，同じ実行をデバッガ上で再現した．記録ファイルのサイズは，4.2MB であった．これは，すべてシステムコールの実行結果の記録によるものである．また実行の記録・再生にかかった時間は，それぞれ 52.7 秒と 44.8 秒であった．

（2）状況の確認　デバッギは，不正な解放を行った時点で停止している．これは，node.c の 710 行目の free() の呼出しである（図 5.6）．そこで，解放を試みた n->wsptr の指す領域の検査を行った[2]．その結果，n->wsptr は，確かにヒープライブラリによって割り当てられた領域であるが，既に解放されていたことがわかった．つまり，二重解放が発生しているのである．

（3）スライシング　次に，n->wsptr の指す領域に対し，スライシングを行うことにした．そのために，まず n->wsptr の指す領域を割り当てた時点まで逆実行した．これには，4.2.4.1 節で紹介した SIC による逆実行を利用した．割当て時の SIC は，(2) で行った検査から取得可能である．逆実行には，60.4 秒かかった．

[2]DbgStar の提供するヒープライブラリ（3.2.1.1 節）には，割り当てたメモリ領域を検査するための機能がある．この機能は，デバッガを通して利用することが可能である．

ここで，さらに逆実行を行う．割当て時のスタックフレームを調査すると，割当てはdo_length() 内から行われていることがわかる．do_length() は，上述のコマンドの length($1) に対応する処理を行う関数である．そこで，4.2.4.2 節で紹介したフレーム識別子を利用し，do_length() の先頭まで逆実行した．逆実行には，44.8 秒かかった．

最後に，スライシングを有効にし，デバッギの実行を再開した．そして，二重解放を行って停止するまで，スライスの計算を行った．スライスの計算には，4.5 秒かかった．

(4) スライスの調査 n->wsptr の指す領域に対するスライスには，ネイティブコードで 64 命令，ソースコードに換算して 28 行が含まれていた．ソースコードの 28 行を調査したところ，wsptr の割当てと解放に直接関係がありそうなのは，図 5.7 の reset_record() と，図 5.6 の str2wstr() であった．それぞれの図において，スライスに含まれる行には S と印をつけた．

reset_record() は，パースしたフィールドをリセットし，空のフィールドで初期化する関数である．reset_record() では，空のフィールドの作成時（296 行目）に，wsptr の値を変更する可能性がある．また str2wstr() は，マルチバイト文字列をワイド文字列に変換する関数である．str2wstr() では，まず wsptr の指す領域を解放する（710 行目）．710 行目は，不正な解放で停止した位置でもある．そして，新たなワイド文字列の領域を wsptr に割り当てる（728 行目）．

(5) スライスに沿った逆実行 最後に，スライスに沿って逆実行を行うことにした．これには，4.2.3 節で紹介したロギング方式の逆実行を利用した．まず，スライスを計算した実行区間において，状態差分の保存を行った．保存には，31.5KB のメモリと 61.0 秒の時間が必要だった．また以降のロギング方式による逆実行は，すべて 0.1 秒以内に完了した．

そして，(4) で述べた 3 行に対しブレークポイントを設定し，逆実行をしていった．すると，str2wstr() の 710 行目で二重解放をした領域は，reset_record() の 296 行目で，Null_field からコピーされたものであることがわかった．さらに逆実行をしていくと，やはり str2wstr() の 710 行目で，全く同じ領域が既に解放されていたことがわかる．この場合も，n->wsptr は，reset_record() の 296 行目でコピーされたものであった．つまり，Null_field の wsptr メンバが，コピーと解放を繰り返されていたことがわかる．

さらに逆実行をしていくと，二重解放した領域の割当て時（str2wstr() の 728 行目）にたどり着く．割当ては，Null_field の wsptr メンバに対して行われていた．ここ

```
      NODE *str2wstr(NODE *n, size_t **ptr)
      {
          ...
709:      if (n->wstptr != NULL) {
710: S        free(n->wstptr);
711:          n->wstptr = NULL;
712:          n->wstlen = 0;
713:      }
          ...
728: S    emalloc(n->wstptr, wchar_t *,
                  sizeof(wchar_t) * (n->stlen + 2),
                  "str2wstr");
729: S    wsp = n->wstptr;
          ...
      }
```

図 5.6 `str2wstr()` 関数（`node.c`）

でスタックフレームを調査すると，`Null_field` への割当ては，入力（`large.txt`）の空行によって発生していた．このことから，不具合は，次のようにして起こったことが判明した．

（i）空行の入力時に，`Null_field->wsptr` に領域が割り当てられてしまう，

（ii）フィールドリセット時に，空のフィールドを初期化するために，`Null_field` がコピーされる．その際，`wsptr` の指す領域のアドレスもコピーされる．

（iii）コピーされたアドレスから，(i）で割り当てた領域が解放される．

（iv）（ii）と（iii）が繰り返される．

5.2.3 Apache HTTP Server

Apache HTTP Server[59]（以降 Apache）は，非常に広く利用されている HTTP サーバである．Apache は，モジュール化された設計を採用しており，それはサーバの基本機能にまで拡張されている．例えば，Apache では，MPM（Multi-Processing Module）を選択することにより，子プロセスやスレッドの管理方法を変更することができる．

MPM には，`prefork` モジュールや `worker` モジュールなどがある．`prefork` は，予め `fork()` で生成しておいた子プロセスを用い，クライアントのリクエストを処理するモジュー

```
      void reset_record()
      {
          ...
293:      for (i = 1; i <= parse_high_water; i++) {
294: S        unref(fields_arr[i]);
295: S        getnode(n);
296: S        *n = *Null_field;
297: S        fields_arr[i] = n;
298:      }
          ...
      }
```

図 5.7 reset_record() 関数（field.c）

ルである．これに対し，worker は，マルチプロセスとマルチスレッドのハイブリッド型のモジュールである．worker では，個々の子プロセスがさらに複数のスレッドを生成しておき，それぞれのスレッドがクライアントのリクエストを処理する．

不具合の概要　Apache 2.0.51 の mod_mem_cache モジュールには，不正なメモリ解放により強制終了する不具合が存在する [39]．Apache は，ローカルのコンテンツや，プロキシされたコンテンツをキャッシュするために，mod_cache モジュールを提供している．mod_mem_cache モジュールは，mod_cache モジュールのサポートモジュールであり，メモリを使用したストレージ管理機構を提供する．mod_mem_cache モジュールの不具合は，MPM に worker モジュールを選択し，キャッシュに大きな負荷をかけた際に発生する．

デバッグシナリオ　デバッガを利用するために，Apache の構築時に，configure スクリプトに対していくつかのオプションを指定した．指定したオプションは，デバッグ情報を生成するためのものと，3.2.1.1 節で述べたライブラリをリンクするためのものである．Apache のソースコードに対しては，特に修正を行う必要はなかった．また DbgStar では，fork() の実行を禁止している．そのため，HTTP サーバの起動時に開発者用のオプションを指定し，worker モジュールが子プロセスを生成しないようにした．

デバッグは，次のステップで行った．

（1）不具合の記録と再現　まず，キャッシュに大きな負荷をかけるために，設定ファイルを修正した．そして，キャッシュするオブジェクトの最大個数を 3 に制限した．次に，HTTP サーバからコンテンツを取得するテストスクリプトを実行した．スクリプト

は，5 つのスレッドを生成し，HTTP サーバに同時にアクセスする．すると，スクリプトがアクセスを 173 回行った時点で，HTTP サーバは不正なメモリ解放によって強制終了された．そこで，記録した実行をデバッガ上で再現した．ここで，記録ファイルのサイズは 5.45MB であった．内訳は，システムコールの実行結果の記録によるものが 5.37MB，スケジューリングの記録によるものが 83.1KB であった．また実行の記録・再生にかかった時間は，それぞれ 3.6 秒と 2.7 秒（ユーザ時間とシステム時間の合計）であった．

（2）状況の確認 デバッギは，不正な解放を行った時点で停止している．デバッギには，20 個以上のスレッドが存在していたが，不正な解放を行ったのはスレッド 17 であった．不正な解放は，`memcache_cache_free()` 内の，`cleanup_cache_object()` の呼出しで発生していた（図 5.8：164 行目）．`memcache_cache_free()` は，キャッシュから溢れて追い出されたオブジェクトを解放するために，呼び出されていた．また `cleanup_cache_object()` は，キャッシュオブジェクトに割り当てられたメモリ領域を解放する関数である．

ここで，DbgStar の提供するヒープライブラリの機能を利用し，`obj` の指す領域の検査を行った．その結果，`obj` は，確かにヒープライブラリによって割り当てられた領域であるが，既に解放されていたことがわかった．つまり，二重解放が発生しているのである．

`mod_mem_cache` モジュールは，このような二重解放を防ぐために，`obj->refcount` と `obj->cleanup` の 2 つのメンバ変数を利用している．`refcount` は，オブジェクトの参照数を保存するメンバである．また `cleanup` は，オブジェクトがキャッシュから削除されている場合に真となるメンバである．図 5.8 に示したように，`memcache_cache_free()` では，最初に `refcount` の値をインクリメントする（159 行目）．次に，`cleanup` の値を真にする（161 行目）．そして，再び `refcount` の値をデクリメントし，結果が 0 になった場合にだけ，`cleanup_cache_object()` を呼び出す（163 行目）．

ここで，`refcount` と `cleanup` の値を調査してみたところ，値はそれぞれ 0 と 1 であった．デバッギは 164 行目で停止しているため，図 5.8 のコードを見る限りでは，それぞれの値に問題はないように見える．

（3）スライシング ここで，`obj->refcount` の値に対し，スライシングを行ってみることにした．そのために，`ap_process_request()` の先頭まで逆実行した．これは，不具合を生じたリクエスト処理の開始地点である．この逆実行には，4.2.4.2 節で紹介したフレーム識別子を利用した．逆実行には，2.6 秒かかった．次にスライシングを有効にし，デバッギの実行を再開した．そして，不正な解放で停止するまで，スラ

イスの計算を行った．スライスの計算には，4.6 秒かかった．

(4) スライスの調査 obj->refcount の値に対するスライスには，ネイティブコードで 1820 命令，ソースコードに換算して 253 行が含まれていた．ただし 4.1 節で述べたように，スライスに含まれる命令はスレッドごとに区別している．そのため，この数字は，同じ命令と行が一部重複して数えられているものである．

ここで，ソースコードの 253 行の中から，mod_mem_cache.c に含まれる 33 行を調査してみた．すると，不正な解放を行ったスレッド 17 ではなく，スレッド 21 が，既に cleanup_cache_object() を呼び出していたことがわかった．この呼出しは，図 5.9 に示した decrement_refcount() から行われていた．decrement_refcount() は，キャッシュオブジェクトの利用終了時に呼び出される関数である．図 5.9 に示したように，decrement_refcount() は，291 行目で refcount（参照数）をデクリメントする．そして refcount が 0 になり，かつ cleanup が真であれば，cleanup_cache_object() を呼び出す．なお，図 5.8 と図 5.9 において，スレッド 17 とスレッド 21 のスライスに含まれる行には，それぞれ S と*という印をつけた．

(5) スライスに沿った実行 （4）の調査から，最初の解放は decrement_refcount() で起こったことがわかった．しかし，図 5.8 と図 5.9 のコードは，refcount と cleanup を利用して，二重解放を防ごうとしているように見える．そこで，デバッギをスライスに沿って実行し，refcount と cleanup がどのように変化していったのか調査することにした．これには，4.2.4.3 節で紹介した再実行方式の逆実行を利用した．まず，スライスを計算した実行区間においてプロファイリングを行い，スライスに含まれる行の実行回数を計測した．プロファイリングには，3.5 秒かかった．

表 5.4 に，プロファイリングの結果を示す．表 5.4 において，タイムインターバルは，単一のスレッドの連続した実行区間（実行権が与えられてから，失われるまで）を表す識別子である．また実行回数は，タイムインターバル中に，スライスに含まれる行が実行された回数である．表 5.4 から，(i) 不正な解放を行ったスレッド 17 は，タイムインターバル 22 でだけ実行されていること，(ii) スレッド 17 の実行は，最初に解放を行ったスレッド 21 の実行に挟まれていることがわかる．そのため，タイムインターバル 22 の前後で，何らかの競合が発生した可能性がある．そこで，タイムインターバル 20 において，スライスに含まれる行が最後に実行された時点まで逆実行した．そして，そこからスライスに沿ってステップ実行をしていった．逆実行には，3.2 秒かかった．また以降のステップ実行は，すべて 0.1 秒以内に完了した．

表 5.5 に，refcount と cleanup の値の変遷を示す．refcount と cleanup の値は，それぞれの行の実行直前の値である．表 5.5 から，不具合は，次のようにして起こっ

```
      static void memcache_cache_free(void*a)
      {
153: S     cache_object_t *obj = (cache_object_t *)a;
           ...
159: S     apr_atomic_inc(&obj->refcount);
160:
161:       obj->cleanup = 1;
162:
163: S     if (!apr_atomic_dec(&obj->refcount)) {
164:           cleanup_cache_object(obj);
165:       }
      }
```

図 5.8 memcache_cache_free() 関数（スレッド 17）

たことが判明した．

（i）スレッド 21 が，decrement_refcount() の 291 行目で，refcount をデクリメントする．その直後に，コンテキストスイッチが発生する．

（ii）スレッド 17 が，memcache_cache_free() の 159 行目で，refcount をインクリメントする．次に，cleanup の値を真にし，163 行目で再び refcount をデクリメントする．その結果，refcount と cleanup の値は，それぞれ 0 と 1 になる．そして 164 行目を実行する前に，コンテキストスイッチが発生する．

（iii）スレッド 21 では，decrement_refcount() の 292 行目の条件が真となるため，293 行目で cleanup_cache_object() を呼び出す．

（iv）その後，スレッド 17 に戻ると，164 行目で再び cleanup_cache_object() が呼び出される．そのため，デバッギは二重解放により強制終了される．

5.2.4 親和性と柔軟性の考察

親和性 本章で紹介したシナリオでは，デバッギ（ProFTPD，GNU Awk，Apache HTTP Server）のソースコードに対して，修正を行う必要は全くなかった．いずれのシナリオでも，configure スクリプトに対していくつかのオプションを指定することにより，本研究で構築したデバッガを利用することができた．このことから，DbgStar は，目標とした既存の開発環境に対する高い親和性を実現できているものと考えられる．

```
      static apr_status_t decrement_refcount(void *arg)
267:  {
268: *     cache_object_t *obj = (cache_object_t *) arg;
          ...
291: *     if (!apr_atomic_dec(&obj->refcount)) {
292:           if (obj->cleanup) {
293: *             cleanup_cache_object(obj);
294:           }
295:       }
296:       return APR_SUCCESS;
      }
```

図 5.9 decrement_refcount() 関数（スレッド 21）

表 5.4 プロファイリング結果

タイムインターバル	スレッド	実行回数
3	23	10
6	28	6
11	21	297
12	23	3
13	21	2
15	21	2
17	21	119
20	21	81
22	17	9
23	21	10
25	21	7
26	21	3
28	21	1

表 5.5 refcount と cleanup の値の変遷

スレッド	関数	行	refcount	cleanup
21	decrement_refcount()	291	1	0
17	memcache_cache_free()	159	0	0
17	memcache_cache_free()	163	1	1
21	decrement_refcount()	293	0	1

表 5.6 シナリオで利用したデバッガの機能

機能		主な監視内容
実行の記録		システムコール，シグナル，スケジューリング
実行の再生		システムコール，シグナル，スケジューリング
スライシング		メモリの読書き，命令の実行
可逆実行（ロギング方式）		メモリの上書き，行・関数の実行
可逆実行（再実行方式）	SIC	SIC のインクリメント
	フレーム識別子	関数の呼出し・復帰
	スライス	命令の実行

柔軟性　本章で紹介したシナリオで利用したデバッガの機能を，表 5.6 に示す．各シナリオでは，表 5.6 に挙げた機能を必要に応じ組み合せながら，デバッグを行っていった．これは，DbgStar の柔軟性に依るところが非常に大きい．表 5.6 に示したように，それぞれの機能では，必要となる監視内容が大きく異なる．DbgStar は，デバッガのユーザが使用する機能に応じ，監視内容を柔軟に変更していくことができた．

このことは，デバッガの機能のオーバーヘッドを限定し，また解析結果を向上させるためにも非常に重要である．例えば表 5.7 に，各シナリオの実行全体に渡って，スライシングのための監視コードを挿入し，計算時間とスライスに含まれる行数を計測した結果を示す．表 5.7 に示したように，Apache HTTP Server のシナリオでは，計算時間とスライスに含まれる行数は，それぞれ 21.2 分と 14K 行であった．また GNU Awk のシナリオでは，計算時間が 10.8 時間にも達する．本章で紹介したシナリオでは，デバッギの実行中に，必要最小限の実行区間に対してだけ，スライシングのための監視コードを挿入した．これにより，計算時間とスライスに含まれる行数を劇的に改善することができた．

以上のことから，DbgStar は，目標とした高い柔軟性を実現できているものと考えられる．

表 5.7 実行全体に渡るスライシング

プログラム	計算時間	スライスに含まれる行数
ProFTPD	39.6 秒	4 行（`digest`）
		208 行（`type`）
GNU Awk	10.8 時間	124 行
Apache HTTP Server	21.2 分	14K 行

第6章　まとめ

本論文では，デバッガの開発者に対し，プログラム実行の制御と監視を行うための基盤環境を提供する DbgStar を提案した．プログラム実行の制御と監視は，デバッガの様々な機能の基礎となる処理である．しかし，既存研究で用いられてきた手法には，オーバーヘッド，親和性，柔軟性の 3 つの要件を同時に満たせるものが存在しなかった．DbgStar では，これらの要件を実現するために，仮想マシンを利用して動的なコード変換を行う手法を，デバッガへと応用した．

本論文では，実際に DbgStar を利用して構築したデバッガを紹介した．デバッガには，プログラムスライシングや可逆実行など，デバッグに有用と思われる様々な高度な機能を統合した．プログラムスライシングは，プログラムのコード間の動的なデータ依存関係を解析する機能である．また可逆実行は，通常のプログラムの実行方向とは，逆向きの実行を可能にする機能である．さらに，これらの機能を用い，オープンソースプログラム（ProFTPD，GNU Awk，Apache HTTP Server）に含まれていた実際の不具合のデバッグを行うシナリオを示した．

そして本論文では，オーバーヘッド，親和性，柔軟性の 3 つの観点から，DbgStar の評価・考察を行った．評価では，DbgStar のオーバーヘッドが，逐次解釈実行の手法に基づいた LVM と比較し，はるかに小さいことを示した．ただし，静的な instrumentation の手法に基づいた ATOM と比較すると，DbgStar のオーバーヘッドの方が大きかった．これは，DbgStar では，仮想マシンを利用して実行時に instrumentation を行っているため，ある程度は仕方のないことである．また，DbgStar のオーバーヘッドには，まだ改善の余地があることも示した．そして，上述のデバッグシナリオでは，デバッギのソースコードに対して修正を行う必要が全くなかったことから，DbgStar が既存の開発環境に対する高い親和性を実現していることを示した．さらに，上述のデバッグシナリオでは，デバッガのユーザが使用する機能に応じ，監視内容を柔軟に変更していくことができたことから，高い柔軟性を実現していることを示した．

今後の課題　今後の課題としては，次のものが挙げられる．

- **対応システムコールの拡充**　3.3.2.1 節で述べたように，DbgStar は，頻繁に利用されるものを中心に，約 70 種類程のシステムコールをサポートしている．しかし，それ以

外のシステムコールへの対応は，今後の課題である．特に fork() など，実行そのものを禁止しているシステムコールへの対応は優先度が高い．fork() をサポートするためには，操作ライブラリを拡張し，親プロセスと全く同様に，複製された子プロセスの実行を制御・監視する仕組みを導入する必要がある．

- **不具合の発生しやすいスケジューリングポリシーの研究** 3.5 節で述べたように，DbgStar では，フラグメントの途中で先取りが起きない．これは，そのような位置で先取りが起きなければ，デバッギの不具合が発生しない場合などに大きな問題となる．しかし，任意の位置での先取りを許可することにも，次の問題がある．

  - 先取りの検査に伴うオーバーヘッドが，非常に大きくなる．
  - あくまで不具合が発生する可能性が生じるだけで，実際に発生させるまでに必要なテストの回数は不定である．

  そのため，効率的に不具合を発生させるための，新たなスケジューリングポリシーの研究が今後の課題である．これには，実際のマルチスレッドプログラムで起こった不具合の調査が必要になると考えられる．

- **実行速度のさらなる向上** 5.1.1 節で述べたように，DbgStar のオーバーヘッドには，まだ改善の余地がある．特に，(i) 仮想マシンにおける変換コードの最適化と，(ii) 最適なスケジューリングポリシーの研究が今後の課題である．(i) に関しては，監視コードにおける不要なレジスタスピルの省略や，イベントハンドラのインライン化などが効果的であると期待される．(ii) に関しては，上述した不具合の発生しやすいスケジューリングポリシーとの兼ね合いになる．いずれにせよ，実際のマルチスレッドプログラムをベンチマークとして，その傾向を調査することが必要になると考えられる．

謝 辞

本研究の機会を与えて下さり，絶えず御指導，御助言を頂いている，高田眞吾 准教授に，深く感謝致します．

また，私が慶應義塾大学理工学部在学中より御指導を頂き，御退職後も絶えず御助言を頂き，さらに忙しい中，本論文のチェックを行って頂いた，中央大学 土居範久 教授に，深く感謝致します．

そして忙しい中，本論文のチェックを行って頂いた，山本喜一 教授，河野健二 准教授，櫻井彰人 教授に，深く感謝の念を表します．

最後に，日頃より共に研究活動に励んできた高田研究室の皆様に，心より感謝致します．

なお，本研究は，日本学術振興会 科学研究費補助金（特別研究員奨励費）の助成を受けたものである．

2008 年 1 月

孝壽俊彦

論文目録

定期刊行誌掲載論文（主論文に関連する原著論文）

[1] 孝壽俊彦，高田眞吾，土居範久．“Dynamic Translation を利用した可逆デバッガ”．ソフトウェア科学会論文誌「コンピュータソフトウェア」，Vol.22，No.3，pp.186-193，2005（日本ソフトウェア科学会 第 11 回論文賞 受賞）．

[2] 孝壽俊彦，高田眞吾，土居範久．“既存の開発環境との互換性と高速な実行を実現したプログラム実行制御・監視環境”．情報処理学会論文誌，Vol.46，No.12，pp.3040-3053，2005.

[3] 孝壽俊彦，高田眞吾，土居範久．“マルチスレッドに対応したプログラム実行制御・監視環境”．情報処理学会論文誌「プログラミング」，Vol.48，No.SIG4，pp.14-26，2007.

定期刊行誌掲載論文（その他の論文）

[1] 孝壽俊彦，高田眞吾，土居範久．“仮想マシンの中間言語に基づく回帰テスト選択手法”．情報処理学会論文誌，Vol.45，No.8，pp.2043-2054，2004.

国際会議論文（査読付きの full-length papers）

[1] Toshihiko Koju, Shingo Takada, and Norihisa Doi. “Regression Test Selection based on Intermediate Code for Virtual Machines”. In Proceedings of the 19th IEEE International Conference on Software Maintenance, pp.420-429, 2003.

[2] Toshihiko Koju, Shingo Takada, and Norihisa Doi. “An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach”. In Proceedings of the First ACM/USENIX Conference on Virtual Execution Environments, pp.79-88, 2005.

国内研究会発表

[1] 孝壽俊彦，高田眞吾，土居範久．“仮想マシンの中間言語に基づく回帰テスト選択手法”．第 10 回ソフトウェア工学の基礎ワークショップ，pp.113-124，2003.

[2] 孝壽俊彦，高田眞吾，土居範久．“Dynamic Translation を利用した可逆デバッガ”．ソフトウェア科学会第 21 回大会，pp.119-123，2004.

[3] 孝壽俊彦，高田眞吾，土居範久．“マルチスレッドに対応したプログラム実行制御・監視環境”．2006 年並列/分散/協調処理に関する『高知』サマー・ワークショップ，2006.

参考文献

[1] H. Agrawal, R. Demillo, and E. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, Vol. 8, No. 3, pp. 21–26, 1991.

[2] H. Agrawal, R. Demillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice & Experience*, Vol. 23, No. 6, pp. 589–616, 1993.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 246–256, 1990.

[4] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. プログラミング言語 AWK. トッパン, 1988. 足立高徳（訳）.

[5] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 18–25, 2002.

[6] T. Akgul and V. J. Mooney. Assembly instruction level reverse execution for debugging. *ACM Transaction on Software Engineering and Methodology*, Vol. 13, No. 2, pp. 149–198, 2004.

[7] T. Akgul, V. J. Mooney, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering*, pp. 522–531, 2004.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 1–12, 2000.

[9] A. Besedes, T. Gergely, Z. M. Szabo, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pp. 105–113, 2001.

[10] B. Boothe. Efficient algorithms for bidirectional debbugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 299–310, 2000.

[11] D. P. Bovet and M. Cesati. 詳解 LINUX カーネル. オライリー・ジャパン, 2001. 高橋浩和，早川仁（監訳），岡島順治郎，三浦広志，田宮まや（訳）.

[12] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 265–275, 2003.

[13] B. R. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, Vol. 14, No. 4, pp. 317–329, 2000.

[14] S. Chen, W. K. Fuchs, and J. Chung. Reversible debugging using program instrumentation. *IEEE Transaction on Software Engineering*, Vol. 27, No. 8, pp. 715–727, 2001.

[15] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 128–137, 1994.

[16] J. J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, Vol. 45, No. 6, pp. 608–619, 2002.

[17] M. L. Corliss, E C. Lewis, and A. Roth. DISE: a programmable macro engine for customizing applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 362–373, 2003.

[18] M. L. Corliss, E C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with DISE. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 303–314, 2005.

[19] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, Vol. 17, No. 3, pp. 36–43, 1997.

[20] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing*, Vol. 11, No. 2, pp. 125–150, 2000.

[21] C. Demetrescu and I. Finocchi. A portable virtual machine for program debugging and directing. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pp. 1524–1530, 2004.

[22] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 297–302, 1984.

[23] Ralf S. Engelschall. GNU Pth - The GNU Portable Threads. `http://www.gnu.org/software/pth/`.

[24] S. I. Feldman and C. B. Brown. IGOR: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 112–123, 1988.

[25] Free Software Foundation, Inc. GAWK: the GNU Awk. `http://www.gnu.org/software/gawk/`.

[26] Free Software Foundation, Inc. GCC: the GNU Compiler Collection. `http://gcc.gnu.org/`.

[27] Free Software Foundation, Inc. GDB: the GNU project debugger. `http://www.gnu.org/software/gdb/`.

[28] Freescale Semiconductor, Inc. 32 ビット PowerPC アーキテクチャ プログラミング環境. 2005.

[29] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation.*, pp. 285–298, 2007.

[30] Intel, Inc. Intel ATOM (Analysis Tools for Object Modification). `http://www.intel.com/cd/software/products/asmo-na/eng/219608.htm`.

[31] Intel, Inc. *Intel Architecture Software Developer's Manual Volume 3: System Programming*. 1999.

[32] Jorn Lind-Nielsen. BuDDy: Binary Decision Diagrams Library Package. `http://sourceforge.net/projects/buddy`.

[33] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pp. 191–206, 2002.

[34] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 66–79, 1994.

[35] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Tdb: a source-level debugger for dynamically translated programs. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, pp. 123–132, 2005.

[36] Naveen Kumar and Ramesh Peri. Transparent debugging of dynamically instrumented programs. In *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications*, pp. 57–62, 2005.

[37] J. R. Levine. *Linkers & Loaders*. オーム社, 2001. 榊原一矢（監訳），ポジティブエッジ（訳）.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pp. 190–200, 2005.

[39] Mailing list archives: dev@httpd.apache.org. Seg fault: Possible race conditions in mod_mem_cache.c. `http://mail-archives.apache.org/mod_mbox/httpd-dev/200409.mbox/browser`.

[40] Mailing lists archives: bug-gnu-utils. Multiple frees in awk, sometimes leading to crash. `http://lists.gnu.org/archive/html/bug-gnu-utils/2007-04/msg00019.html`.

[41] K. Maruyama and M. Terada. Timestamp based execution control for C and Java programs. In *Proceedings of Workshop on Automated and Algorithmic Debugging*, pp. 87–102, 2003.

[42] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 78–86, 1989.

[43] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, Vol. 28, No. 11, pp. 37–46, 1995.

[44] T. Moher. PROVIDE: a process visualization and debugging environment. *IEEE Transaction on Software Engineering*, Vol. 14, No. 6, pp. 849–857, 1988.

[45] N. Nethercote and J. Seward. Valgrind: a program supervision framework. *Electronic Notes in Theoretical Computer Science*, Vol. 89, No. 2, 2003.

[46] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pp. 89–100, 2007.

[47] Bradford Nicbols, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads プログラミング. オライリー・ジャパン, 1998. 榊正憲（訳）.

[48] J. B. Rosenberg. デバッガの理論と実装. アスキー出版局, 1998. 吉川邦夫（訳）.

[49] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pp. 258–286, 1996.

[50] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 209–218, 2002.

[51] K. Scott, J. Davidson, and K. Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, University of Virginia, 2001.

[52] K. Scott, N. S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 36–47, 2003.

[53] R. Sosic. Dynascope: a tool for program directing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 12–21, 1992.

[54] R. Sosic. Design and implementation of Dynascope, a directing platform for compiled programs. *Computing Systems*, Vol. 8, No. 2, pp. 107–134, 1995.

[55] A. Srivastava and D. W. Wall. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 196–205, 1994.

[56] Standard Performance Evaluation Corporation. SPEC CPU2000. `http://www.spec.org/cpu2000/`.

[57] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 117–130, 1999.

[58] Thai Open Source Software Center Ltd and Clark Cooper. The Expat XML Parser. `http://expat.sourceforge.net/`.

[59] The Apache Software Foundation. The Apache HTTPD Server. `http://httpd.apache.org/`.

[60] The Open Group. The Single UNIX Specification, version 2, 1997. `http://www.opengroup.org/onlinepubs/007908799/index.html`.

[61] The ProFTPD Project. Bugzilla Bug 2658 - Segfault in mod_radius when using long password. `http://bugs.proftpd.org/show_bug.cgi?id=2658`.

[62] The ProFTPD Project. ProFTPD. `http://www.proftpd.org/`.

[63] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, Vol. 3, No. 3, pp. 121–189, 1995.

[64] Valgrind Developers. Valgrind. `http://valgrind.org/`.

[65] R. Wahbe. Efficient data breakpoints. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 200–212, 1992.

[66] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 1–12, 1993.

[67] M. Weiser. Program slicing. In *Proceedings of of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[68] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of Conference on Measurement and Modeling of Computer Systems*, pp. 68–79, 1996.

[69] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, 1995.

[70] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 2, pp. 1–36, 2005.

[71] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 319–329, 2003.

[72] X. Zhang and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the 26th International Conference on Software Engineering*, pp. 502–511, 2004.

[73] 丸山一貴, 寺田実. 関数単位疑似逆実行の高速化. 情報処理学会論文誌「プログラミング」, Vol. 41, No. SIG9, pp. 1–7, 2000.

[74] 荒木啓二郎, 張漢明. IT Text プログラム仕様記述論. オーム社, 2002.

[75] 柳澤佳里, 光来健一, 千葉滋, 石川零. OS カーネル用アスペクト指向システム KLASY. 情報処理学会論文誌「プログラミング」, Vol. 48, No. SIG10, pp. 176–188, 2007.

付 録A　コード変換情報

本研究では，DbgStar の仮想マシンと操作ライブラリの間で，仮想マシンが行ったコード変換の詳細に関する情報をやり取りするために，コード変換情報を定義した．以下では，主なコード変換情報として，EMIT，TRANS，LINK，BP の 4 種類を紹介する．

A.1　EMIT

EMIT は，キャッシュに出力された個々のフラグメントに関する情報である．EMIT に含まれるフィールドは，次の通りである．

（1）`orig_address`**（4 bytes）** オリジナルフラグメントの開始アドレス

（2）`orig_size`**（4 bytes）** オリジナルフラグメントのサイズ

（3）`address`**（4 bytes）** 変換済みフラグメントの開始アドレス

（4）`size`**（4 bytes）** 変換済みフラグメントのサイズ

A.2　TRANS

TRANS は，単一の命令が，複数の命令からなるコードにコード変換されたことを表す情報である．TRANS に含まれるフィールドは，次の通りである．

（1）`orig_address`**（4 bytes）** オリジナル命令の開始アドレス

（2）`orig_size`**（4 bytes）** オリジナル命令のサイズ

（3）`address`**（4 bytes）** 変換コードの開始アドレス（相対）

（4）`size`**（4 bytes）** 変換コードのサイズ

（5）`num_last`**（4 bytes）** 末尾命令の個数

（6）`lasts`**（**`num_last` × **4 bytes）** 末尾命令のアドレス（相対）

`lasts` は，ステップ実行で利用するためのものである（3.4.1.2 節）．また，`address` と `lasts` は，変換済みフラグメントの開始アドレスからの相対アドレスである．

A.3 LINK

LINK は，変換済みフラグメントのリンキングにより，末尾命令のアドレスが変更されたことを表す情報である（3.3.1 節）．LINK に含まれるフィールドは，次の通りである．

（1）`last`**（4 bytes）** 新しい末尾命令のアドレス

A.4 BP

BP は，仮想マシンがコード変換時に設定したブレークポイントに関する情報である（3.4.1.2 節）．BP に含まれるフィールドは，次の通りである．

（1）`orig_address`**（4 bytes）** ブレークポイントを設定したオリジナル命令のアドレス

（1）`address`**（4 bytes）** ブレークポイントを設定した変換コードのアドレス

（1）`code`**（4 bytes）** ブレークポイント命令で置き換えたコード