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

孝壽俊彦[†] 高田眞吾[†] 土居範久[†,††]

ソフトウェアを開発する際には，プログラムの実行時の振舞いを調査するためのツール群が必要不可欠となる．このようなツールは director と呼ばれている．Director の開発に必要な労力を軽減するために，従来からプログラム実行制御・監視環境が提案されている．しかし従来のプログラム実行制御・監視環境では，既存の開発環境との互換性と実行速度の面で大きな問題があった．本論文では，これらの問題を解決することができる新しいプログラム実行制御・監視環境を提案する．提案環境では，機械語プログラムの実行の制御と監視を行うことにより，互換性の問題を解決する．また Dynamic Translation を行い，対象プログラムと監視コードの実行を機械語コードによって直接行うことにより，実行速度の問題を大幅に改善する．

An Efficient Directing Platform Compatible with Existing Development Environments

Toshihiko Koju,[†] Shingo Takada[†] and Norihisa Doi[†,††]

Software developers need sets of tools that can investigate program behavior. These tools are called directors. The development of directors is very costly. So, directing platforms have been proposed to support the development of new directors. But existing directing platforms have two serious problems: (1) compatibility with existing development environments, and (2) efficiency in terms of execution speed. In this paper, we propose a new directing platform that can solve these problems. We solve the compatibility problem by using native machine code as the target of our directing platform. In addition, we greatly improve execution speed by dynamically translating the target program to include monitoring code, and executing both the target program and monitoring code directly on the real CPU.

1. 序　　論

ソフトウェアを開発する際には，プログラムの実行時の振舞いを調査するためのツール群が必要不可欠となる．これは，実際にプログラムを実行し，その振舞いを調査することが，プログラムの不具合の検出やその原因の特定，開発者のプログラムに対する理解の促進などに欠かすことができないためである．たとえばエラーチェッカ[8] は，プログラムの実行を自動的に解析し，特定の種類の不具合を検出する目的で用いる．デバッガ[14] は，開発者がプログラムの実行を追跡し，検出された不具合の原因を特定する目的で用いる．ビジュアライザ[5] は，開発者が理解しやすいように，プログラムの挙動をグラフィカルに視覚化する目的で用いる．プロファイラ[2] は，プログラムの性能を計測し，そのボトルネックを発見する目的で用いる．このようなプログラムの実行時の振舞いを調査するためのツールは，*director* [7),16)] と呼ばれる．ソフトウェアがより大規模かつ複雑になるにつれ，director の重要性もますます高まる．

Director は，その種類は違っていても，プログラムの実行の**制御**と**監視**の 2 種類の基本機能を持つ．プログラムの実行の制御と監視は互いに密接に関連した機能であり，これらの機能はまとめて *directing* と呼ばれる．しかし C 言語のプログラムを対象とした directing の実装には，大変な労力が必要となる．これは，C 言語のプログラムの実行には，ハードウェアアーキテクチャ，オペレーティングシステム，そのプログラムの開発環境（e.g. 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．

文献 7) や 16) は，C 言語のプログラムを対象とした director を開発する際に基盤となる *Directing*

† 慶應義塾大学
Keio University
†† 中央大学
Chuo University

Platform を提案している．Directing Platform は，director がプログラムの実行の制御や監視を行うために必要な機能を提供する．そこで本論文では，Directing Platform のことを，**プログラム実行制御・監視環境**と呼ぶ．Director の開発者はプログラム実行制御・監視環境を利用することにより，開発に必要な労力を大幅に軽減することができる．

しかし，文献 7) や 16) で提案されている従来のプログラム実行制御・監視環境では，(1) 既存の開発環境との互換性と，(2) 実行速度の面で大きな問題がある．本論文では，これらの問題を解決することができる新しいプログラム実行制御・監視環境を提案する．

以下，2 章で，従来のプログラム実行制御・監視環境について述べる．3 章で，本論文で提案する新しいプログラム実行制御・監視環境について述べる．4 章で，提案環境の実装について述べる．5 章で，提案環境を利用して実際に構築した director の例として，可逆デバッガを紹介する．6 章で，提案環境の評価について述べる．7 章で，関連研究について述べる．最後に 8 章で，結論と今後の課題について述べる．

2. プログラム実行制御・監視環境

Director は，プログラムの実行の制御と監視の 2 種類の基本機能を持つ：

プログラムの実行の制御 プログラムの実行の管理や，実行時の振舞いを操作するための機能．本論文では，プログラムの実行の開始や終了，停止や継続，状態の取得や修正などの機能を，まとめてプログラムの実行の制御と呼ぶ．

プログラムの実行の監視 プログラムの実行時の振舞いに関する情報を収集するための機能．プログラムは実行状態の変化，例外の発生など，実行時に様々な挙動を示す．このような挙動は，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の捕捉や，イベントに関連した情報を収集する機能を，プログラムの実行の監視と呼ぶ．

プログラム実行制御・監視環境は，director がプログラムの実行の制御や監視を行うために必要な機能を提供する．以下では，従来のプログラム実行制御・監視環境について簡単に紹介し，その後で従来の環境の問題点を述べる．

2.1 Dynascope

Dynascope[16] は，機械語にコンパイルされたプログラムの実行の制御と監視を行うための機能を提供する．Dynascope では，実行の監視を行う際に，対象の機械語プログラムをエミュレータ上で実行する．このエミュレータは，対象プログラムの状態（レジスタやメモリの内容など）の変化を，イベントとして検出することができる．Director は，必要とするイベントに関連した情報のみをエミュレータから受け取ることが可能である．Dynascope のエミュレータは，対象プログラムをインタプリタ方式で実行する．

2.2 Leonardo Virtual Machine

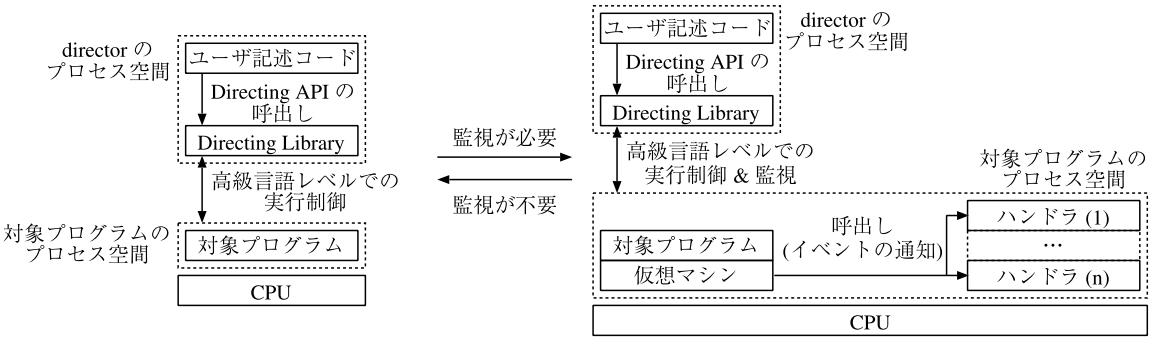
Leonardo Virtual Machine（LVM）[7] は，特殊な中間言語にコンパイルされたプログラムの実行の制御と監視を行うための機能を提供する．この中間言語のコードは，Leonardo C Compiler を利用して生成される．LVM は，基本的な実行の制御を行う機能に加えて，プログラムの逆実行[3),4)] を行う機能を提供している．また LVM は，対象プログラム内部のイベント（例外の発生，関数の呼び出しと復帰，メモリの読み書きなど）から，LVM 内部のイベント（メモリの割当てと解放，プログラムの開始と終了など）まで，様々な種類のイベントを検出することができる．Director は，特定のイベントが発生するまで待機することや，特定のイベントに対して独自のハンドラを登録することが可能である．LVM は，中間言語のコードをインタプリタ方式で実行する．

2.3 従来の環境の問題点

LVM には，既存の開発環境との互換性の面で大きな問題がある．これは，LVM が特殊な中間言語のみを対象としているため，既存の開発環境のライブラリ群（機械語）がそのまま利用できないためである．そのため，LVM を利用して開発された director を利用するためには，まず director の対象プログラムを LVM に移植する手間が必要になる可能性がある．これは，その director の有用性を大きく損ねることにつながる．

もちろん LVM を拡張して，既存のライブラリを呼び出すための機構を用意することも可能である．たとえば，既存のライブラリのコードは，仮想マシンを用いずに CPU 上で直接実行するのである．しかしこの手法では，仮想マシンからライブラリのコードを監視することができない．Director の提供する機能によっては，対象プログラムのみではなく，使用しているライブラリも含めたすべてのコードを監視する必要がある．そのため，このことが非常に大きな問題となる場合がある．そのような機能の例としては，5.1 節で取り上げるヒープ検査などがあげられる．

また Dynascope も LVM も，実行速度の面で大きな問題がある．これは，どちらの環境でも，対象プログラムをインタプリタ方式で実行するためである．



(a) 通常実行モード
(仮想マシンとハンドラは省略)

(b) 仮想実行モード

図 1 提案環境の概要
Fig. 1 Overview of our proposed platform.

C 言語のプログラムは，通常機械語にコンパイルされ，CPU 上で直接実行される．そのようなプログラムをインタプリタ方式で実行するため，その実行速度の低下率は非常に大きなものとなる．

3. 提 案 環 境

2 章で述べたように，従来のプログラム実行制御・監視環境では，(1) 既存の開発環境との互換性と，(2) 実行速度の面で大きな問題がある．本章では，これらの問題を解決することができる，新しいプログラム実行制御・監視環境を提案する：

(1) 既存の開発環境との互換性 LVM には，既存の開発環境との互換性の面で大きな問題がある．これは，LVM が特殊な中間言語を利用するためである．そこで提案環境では特殊な中間言語の利用は避け，Dynascope のように，機械語にコンパイルされたプログラムの実行の制御と監視を行う．これにより，既存の開発環境が提供するコンパイラ，リンカ，ライブラリ群などが，そのまま利用可能となり，既存の開発環境に対する完全な互換性を実現できる．

(2) 実行速度 Dynascope と LVM には，実行速度の面で大きな問題がある．これは，director の対象プログラムをインタプリタ方式で実行するためである．そこで提案環境の仮想マシンでは，Dynamic Translation [1] を行う．Dynamic Translation とは，実行時に仮想マシン内でコード変換を行う手法である．提案環境の仮想マシンでは，対象プログラムのコードに対し，監視用のコードを挿入する．この変換によって生成されるコードも，対象プログラムと同様に機械語コードである．そして仮想マシンは，生成された機械語コードを CPU 上で直接実行する．これにより，対象プログラムと監視コードの実行を，機械語コードによって直接行うことが可能となり，非常に高速な実行を実現できる．

以下では，提案環境の概要と主な機能を紹介する．

3.1 提案環境の概要

提案環境は director に対し，Directing Library と仮想マシンの 2 つのコンポーネントを提供する．Directing Library は，Directing API を通して，実行の制御と監視を行うための機能を director に提供する役割を持つ．仮想マシンは，実際に対象プログラムの実行を監視する役割を持つ．**図 1** に，提案環境において director が実行の制御と監視を行う様子を示す．

図 1 に示したように，提案環境では，director と対象プログラムは別々のプロセスとして実行される．そして director は，つねに Directing Library を通して，対象プログラムのプロセス空間にアクセスする．7 章で述べるように，director と対象プログラムを別々のプロセスとして実行することは，マルチプロセスに対応した director の構築を支援するために非常に重要である．

上述したように，提案環境では，既存の開発環境でコンパイルされた機械語のプログラムを，そのまま director の対象プログラムとして利用する．対象プログラムに対してあらかじめ必要な準備は，デバッグ情報を有効にしてコンパイルしておくことのみである．これにより，既存の開発環境に対する完全な互換性を実現する．加えて Directing Library は，対象プログラムのデバッグ情報を利用して，機械語プログラムの実行を元のソースコードと対応付けるための機能も提供している．そのため提案環境では，機械語プログラムの実行の制御と監視を高級言語レベルで行うことも

表 1 Directing API の主な機能
Table 1 Main functionalities of Directing API.

実行の制御に関する機能		
(a)	起動とアタッチ 強制終了とデタッチ	提案環境下でプログラムを起動，または提案環境外で実行中のプログラムに接続する． 提案環境下で実行中のプログラムを強制終了，または提案環境外に分離する．
(b)	ブレークポイントと継続実行 ステップ実行 レジスタやメモリの読み書き スコープ情報や型情報の取得	停止位置を設定し，実行を再開する． 機械語コードの 1 命令 or ソースコードの 1 行だけ，実行を進める． レジスタやメモリの内容の取得や修正を行う． ソースコードにおける変数のスコープ情報や，型情報などを取得する．
(c)	逆ステップ実行 逆実行 状態履歴の調査	ソースコードの 1 行 or 1 関数だけ，実行を巻き戻す． 指定されたタイムスタンプまで，実行を巻き戻す． 状態の変更履歴を調査する．
実行の監視に関する機能		
(d)	ハンドラのロードとアンロード ハンドラの登録・抹消	ハンドラを含むモジュールをロード，またはアンロードする． イベントに対してハンドラを登録，または登録を抹消する．
(e)	逆実行の切替え	逆実行の有効化，または無効化を行う．

可能である．

提案環境では，実行の監視が不要な場合には，対象プログラムを CPU 上で直接実行する（図 1 (a)）．本論文では，これを通常実行モードと呼ぶ．そして実行の監視が必要な場合にのみ，対象プログラムを仮想マシン上で実行する（図 1 (b)）．本論文では，これを仮想実行モードと呼ぶ．上述したように，提案環境の仮想マシンは Dynamic Translation を行うことにより，非常に高速な実行を実現する．しかし，それでもなお，対象プログラムを仮想マシン上で実行した場合には，幾分かのオーバヘッドが発生することになる．そのため提案環境では，通常実行モードと仮想実行モードを，必要に応じて切り替える機能を提供している．

3.1.1 通常実行モード

通常実行モードでは，対象プログラムは CPU 上で直接実行される（図 1 (a) ☆）．そのため，Directing API の呼び出しのコストは別途必要となるが，対象プログラムそのものの実行に関しては，まったくオーバヘッドが発生しない．ただし director が利用できる機能も，基本的なもののみに限られる（**表 1** (b)）．

3.1.2 仮想実行モード

仮想実行モードでは，対象プログラムは仮想マシン上で実行される（図 1 (b)）．提案環境では，仮想マシン上での実行の詳細は，Directing Library が内部に隠蔽する．そのため director の開発者は，仮想マシンについてほとんど意識せずに，実行の制御と監視を行う機能を利用できる．たとえば，通常実行モードで利用できる機能は，仮想実行モードでも完全に同一の API を通じて利用可能である（表 1 (b)）．そのため director の開発者は，それぞれのモードに対して別々のコードを記述することなく，これらの機能を利用できる．

さらに仮想実行モードでは，プログラムの逆実行[3),4)]に関連した機能を利用できる（表 1 (c)，(e)）．プログラムの逆実行とは，実行中のプログラムの状態を，過去の時点での状態に巻き戻す機能のことである．いい換えれば，直前に実行したコードが行った操作を，順番にアンドゥしていく機能のことである．4.1.2 項と，4.2.3 項で述べるように，提案環境では，対象プログラムの状態の履歴を仮想マシンが記録することによって，プログラムの逆実行を実現している．

また仮想実行モードでは，実行の監視を行うための機能も利用できる（表 1 (d)）．提案環境の仮想マシンは，対象プログラムの示す様々な挙動をイベントとして検出することができる．Directing Library は，これらのイベントに対して，director が独自のハンドラを登録するための機能を提供している．図 1 (b) に示したように，director が登録したハンドラは，イベントが発生した際に仮想マシンによって自動的に呼び出される．そのため director は，ハンドラを通じてイベントに関連した情報を収集することができる．

3.2 提案環境の主な機能

本節では，director の開発者から見た提案環境について，簡単に紹介する．表 1 に，Directing API の主な機能を示す．

3.2.1 実行の制御に関する機能

表 1 に基づき，実行の制御に関する機能は，次の 3 種類に分類できる：

(a) これらは，提案環境下での実行を管理するための機能である．Director は “起動” か “アタッチ” を行うことで，実行の制御と監視を開始する．そ

☆ 簡単のため，仮想マシンとハンドラは省略している．これは，通常実行モードではまったく利用されないためである．しかし実際には，対象プログラムのプロセス空間に存在している．

表 2 提案環境で監視可能なイベント
Table 2 Available events in our proposed platform.

イベント	説明
LINE_{pre}，PROC_{pre}	ソースコードにおける行や関数の先頭
LOAD_{pre}	メモリの内容の読み込み
$\mathrm{STORE}_{pre,post}$	メモリの内容の上書き
CALL_{pre}，RET_{pre}	関数の呼び出しと復帰
BRANCH_{pre}	関数内での実行の分岐
$\mathrm{SYSCALL}_{pre,post}$	システムコールの呼び出し
CALL_{rep}	関数をハンドラに置換

して “強制終了” か “デタッチ” を行うことで，実行の制御と監視を終了する．“アタッチ” と “デタッチ” は，すでに提案環境外で実行中のプログラムに対して，途中から実行の制御と監視を行うための機能である．

(b) これらは，基本的な実行の制御を行うための機能である．これらの機能は，機械語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．

(c) これらは，プログラムの逆実行を行うための機能である．これらの機能を利用するためには，あらかじめ（e）の機能を用い，逆実行を有効にしておく必要がある．“逆ステップ実行” は， ソースコードの 1 行，または 1 関数だけ，実行を巻き戻すための機能である．また “状態履歴の調査” は，仮想マシンが記録した状態の変更履歴を調査するための機能である．たとえば，指定された変数が最後に変更された時点のタイムスタンプなどの情報を取得することが可能である．そして “逆実行” の機能を用いることで，そのタイムスタンプまで実行を巻き戻すことができる．

3.2.2 実行の監視に関する機能

表 1 に基づき，実行の監視に関する機能は，次の 2 種類に分類できる：

(d) これらは，ハンドラを管理するための機能である．**表 2** に，提案環境で監視可能なイベントの一覧を示す．*pre* と *post* は，それぞれイベントの直前と直後にハンドラが呼び出されることを表している．たとえば LINE_{pre} と PROC_{pre} 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機械語命令の実行直前に，登録されたハンドラが呼び出されるイベントである．また CALL_{rep} は特殊なイベントであり，指定された関数が呼び出された際に，その関数の実行は行わずにハンドラの呼び出しのみを行う．これは，特定の関数をハンドラで置換するためのものである．さらにイベントによっては，追加の条件を指定できるものがある．たとえば LOAD_{pre} や $\mathrm{STORE}_{pre,post}$ 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．

(e) これらは，逆実行の有効，無効を切り替えるための機能である．逆実行を有効にすると，対象プログラムの状態の履歴が，仮想マシンによって記録されるようになる．

図 1 に示したように，提案環境では，director と対象プログラムは別々のプロセスとして実行される．そのため，director はハンドラと**通信**を行い，ハンドラが収集した情報を対象プログラムのプロセス空間から取り出す必要がある．提案環境では，表 1 (b) の機能を利用して，これを実現している．提案環境では，ハンドラを含むモジュールに対しても，高級言語レベルでアクセスすることができる．そのため，収集した情報をモジュール内の適切なデータ構造に保存しておくことにより，director はそれらを自由に取り出して利用することが可能となる．

また提案環境では，1 つでもハンドラが登録されているか，逆実行が有効にされている場合にのみ，仮想実行モードで実行を行う．そしてそれ以外の場合には，通常実行モードで実行を行う．Director が表 1 (d)，(e) の機能を利用すると，Directing Library はこのモードの切替えを自動的に行う．そのため director の開発者は，モードの切替えを特に意識する必要はない．

4. 提案環境の実装

本章では，提案環境の実装について述べる．提案環境は，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対象としており，Intel x86 アーキテクチャの Linux 上で動作する☆．

提案環境の仮想マシンは，i386 および i486 互換の命令セットの中で，特に C 言語のコンパイラによって出力されると思わ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 互換の命令セットは，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 などのコンパイラでは，オプションなどを通して，これらの命令セットのみを出力するように指定することが可能である．

なお，インラインアセンブラなどの機能を利用し，提案環境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，提案環境を利用することができない．また，実行時にコードの変更や生成を行うような自己書換え型のプログラムに対しても，

☆ ただしマルチスレッドに関しては，現在対応策を検討中である．

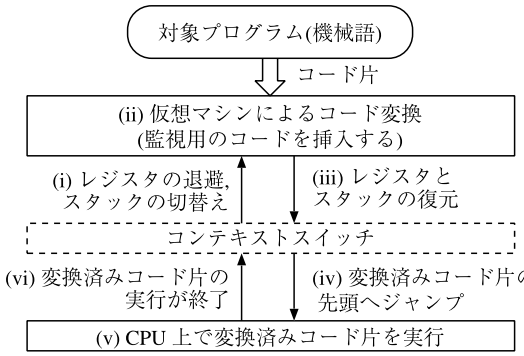


図 2 Dynamic Translation を利用した実行
Fig. 2 Execution using Dynamic Translation.

提案環境を利用することができない．しかしこれらのテクニックは，提案環境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ため，実用上は大きな問題にならないと考えられる．

以下では，まず仮想マシンの実装について述べ，それから Directing Library の実装について述べる．また最後に，提案環境の現在の実装上の制約についても述べる．

4.1 仮想マシン

提案環境の仮想マシンは，Dynamic Translation を行う．**図 2** に，提案環境の仮想マシンが Dynamic Translation を行いながら，対象プログラムを実行していく様子を示す：

(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) 対象プログラムから次に実行すべきコード片を抽出し，監視用のコードを挿入する．
(iii) レジスタとスタック領域を復元する．
(iv) 変換済みコード片の先頭にジャンプする．
(v) 変換済みコード片を CPU 上で直接実行する．
(vi) 変換済みコード片の実行終了後，(i) に戻る．

以下では，提案環境の仮想マシンがステップ（ii）で行うコード変換について，より詳細に述べる．

4.1.1 コード変換

図 3 に，提案環境の仮想マシンが行うコード変換の概要を示す．仮想マシンは，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を変換の単位とする．仮想マシンは，基本的に以下の 2 種類の変換を行う：

監視コードの挿入 監視を行う必要がある命令の前後に，監視コードを挿入する．監視コードには，ハンドラの呼び出しを行うためのものと，逆実行に必要な状態の履歴を保存するためのものがある．図 3 の例では，命令 2 のみ監視を行う必要があるものとし，その直前に監視コードを挿入している．

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変更する．その際，仮想マシンには，本来の分岐先が次の変換開始位置として渡されるようにする．これは，4.1 節のステップ（vi）にあたる．図 3 の命令 4 に，この変換の例を示す．

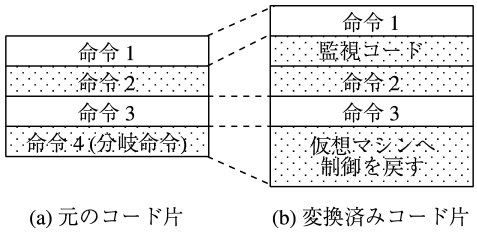


図 3 仮想マシンが行うコード変換の概要
Fig. 3 Overview of code translation.

ただし call 命令と ret 命令を変換する際には，変換済みコードではなく，変換前のコードを指す**本来のリターンアドレス**をスタックへ積むように注意する必要がある．これは，通常実行モードと仮想実行モードとの間でスタックを共有し，切替えを簡略化するためである．そこで call 命令と ret 命令に関しては，まず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変換する☆．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．

また提案環境の仮想マシンでは，実行速度を向上させるために，変換済みコード片のキャッシングとリンキング[15] を行う．キャッシングとは，一度変換したコード片を仮想マシン内のキャッシュに登録する手法のことである．これにより，同じコード片が何度も再変換されることを避けることができる．またリンキングとは，可能なら，キャッシュ内の変換済みコード片の間を，分岐命令で直接結んでしまう手法のことである．これにより，分岐命令で結ばれた変換済みコード片の間では，図 2 のステップ（vi), (i), (ii), (iii), (iv) を省略することができる．

4.1.2 状態の履歴の保存

提案環境では，状態の変更履歴を記録する手法[3),4)] を用いて，プログラムの逆実行を実現している．この手法では，プログラムを通常方向に実行するときに，

☆ その際，仮想マシンには，本来の呼び出し先，もしくはリターンアドレスが，次の変換開始位置として渡されるようにする．

状態（メモリの内容など）の変更を検出し，変更前後での状態の差分（変更前のメモリの内容など）を保存しておく．そしてプログラムの逆実行は，この状態の差分を利用して，変更前の状態を順番に復元していくことによって実現するのである．提案環境では，4.1.1 項で述べた仮想マシンが挿入する監視コードを利用して，状態の変更履歴の保存を行う．

プログラムの状態には，レジスタやメモリの内容からなる内部的な状態と，ファイルなどからなる外部的な状態がある．提案環境では，メモリの内容と外部的な状態に関しては，それらを変更する機械語命令の直前に，状態の差分を保存する監視コードを挿入する☆．一方，レジスタは非常に頻繁に変更されるため，変更のたびに状態の差分を保存すると，深刻なオーバヘッドを発生する恐れがある．そこで提案環境では，レジスタに関しては，ブロックごとにまとめて一度だけ保存を行う監視コードを挿入する．そして提案環境における逆実行の単位を，機械語命令単位ではなく，このブロック単位としている．提案環境では，ブロックとして，C 言語のソースコードにおける行，または関数が選択可能である．

実際に，状態の変更履歴を利用する方法については，4.2.3 項で取り上げる．

4.2 Directing Library

本節では，表 1 にあげた Directing API の機能に沿って，Directing Library の実装について述べる．Directing Library は，通常実行モードでも，仮想実行モードでも，基本的に Unix の Ptrace システムコールや Proc ファイルシステムを用いて，対象プログラムや仮想マシンの操作を行う．

Directing API の機能の中で，表 1 (a)，(b) の機能の一部は，従来のソースレベルデバッガ（GDB など）とよく似た方法で実装されている．そのため以下では，そのような機能の実装については，従来のデバッガとの相違点を中心に述べる．

4.2.1 実行の管理（表 1 (a)）

実行の管理の機能は，基本的に従来のデバッガとほとんど同じ方法で実装されている．ただし Directing Library では，いくつか追加の処理を行う必要がある．まず “起動” と “アタッチ” の機能では，提案環境の仮想マシンを，対象プログラムのプロセス空間にロード（注入）する必要がある．同様に “強制終了” と “デタッチ” の機能では，対象プログラムのプロセス空間から，仮想マシンをアンロードする必要がある．

☆ 4.3 節で述べるように，逆実行可能なシステムコールは，メモリ操作やファイルの入出力など，非常に頻繁に使われているものに限られている．ファイルの入出力に関しては，文献 4) と同様の手法を利用している．

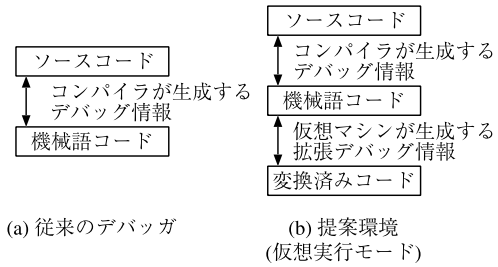


図 4 拡張デバッグ情報
Fig. 4 Extra debugging information.

Directing Library は，環境変数 *LD_PRELOAD* と非常に小さなスタブライブラリを利用して，これらの処理を行う．*LD_PRELOAD* に設定されたライブラリは，新しくプログラムが開始される際に，システムによって自動的にそのプロセス空間にロードされる．そこで director のユーザは，*LD_PRELOAD* にあらかじめスタブライブラリのパスを設定しておくものとする．このスタブライブラリには，仮想マシンのロードとアンロードを行う関数のみが含まれている．

そして Directing Library は，director の対象プログラム自身に，システムによってロードされたスタブライブラリの関数を実行させる．対象プログラムに指定した関数を実行させる機能は，従来のデバッガでも提供されている一般的な機能である．これにより，対象プログラムのプロセス空間に対する，外部からの仮想マシンのロードとアンロードを実現することができる．

4.2.2 基本的な実行の制御（表 1 (b)）

3.1 節で述べたように，基本的な実行の制御の機能は，通常実行モードでも仮想実行モードでも利用することができる．通常実行モードでは，これらの機能は従来のデバッガとまったく同じように実装されている．しかし仮想実行モードでは，従来のデバッガとは異なり，仮想マシンと変換済みコードを扱う必要があるために，同様の方法を利用することはできない．そのため提案環境では，**拡張デバッグ情報**を導入している．

従来のデバッガでは，コンパイラが生成するデバッグ情報のみが利用される．このデバッグ情報には，ソースコードと機械語コードとの間の対応関係が含まれている（**図 4** (a)）．さらに提案環境の仮想実行モードでは，仮想マシンが生成する拡張デバッグ情報を利用する．この拡張デバッグ情報には，機械語コードと変換済みコードとの間の対応関係が含まれている（図 4 (b)）．

Directing Library は，まず仮想マシンの生成する

拡張デバッグ情報を利用することにより，表 1 (b) の機能を元の機械語コードレベルで実現している（ソースコードの情報を必要とするものは除く）．ここで元の機械語コードレベルとは，利用者が変換済みコードを意識することなく，まるで元の機械語コードを対象としているかのように，これらの機能（e.g. ブレークポイントの設定）を利用できることを意味している．

さらに Directing Library は，このように実現した元の機械語コードレベルでの機能と，コンパイラの生成する従来のデバッグ情報を組み合わせることにより，表 1 (b) の機能をソースコードレベルで実現している．この部分に関しては，従来のソースコードレベルデバッガと非常によく似た方法で実装されている．

4.2.3 逆実行（表 1 (c)）

プログラムの逆実行自体の実装は，非常にシンプルである．すなわち，4.1.2 項で述べた監視コードによって保存された状態の変更履歴を読み出し，順番に変更前の状態へと復元していくだけである．

また提案環境は，逆実行を支援するための機能として，状態の変更履歴を調査するための機能も提供している．この機能では，まず director が特定のブロックの先頭（行 or 関数），メモリ領域の変更，システムコールといった検索条件を指定する．すると提案環境は，条件に一致するまで，状態履歴を検索していく．そして条件に一致した場合には，その時点でのタイムスタンプや指定されたメモリ領域の内容などの情報を返す．Director は，ここで取得したタイムスタンプを，逆実行の目的地として利用可能である．

4.2.4 ハンドラの管理と逆実行の切替え（表 1 (d)，(e)）

ハンドラの管理と逆実行の切替えの機能は重複する部分も多いため，本項でまとめて述べる．

4.2.4.1 ハンドラのロードとアンロード

提案環境の仮想マシンのモジュール内には，ハンドラを含むモジュールのロードとアンロードを行うための関数が用意されている．そこで Directing Library は，対象プログラム自身にこれらの関数を実行させることにより，外部からハンドラを含むモジュールのロードとアンロードを行う機能を実現している．

4.2.4.2 ハンドラの登録・抹消と逆実行の切替え

ハンドラの登録・抹消と逆実行の切替えの機能は，以下の 2 種類の操作を組み合わせて実現されている：

通常実行モードと仮想実行モードの切替え 3.2.2 項で述べたとおり，提案環境では，1 つでもハンドラが登録されているか，逆実行が有効にされている場合には，仮想実行モードで実行を行う．そしてそれ以外の場合には，通常実行モードで実行を行う．そのためハンドラの登録・抹消や逆実行の切替えを行う際に，モードの切替えが必要になる場合がある．

提案環境の仮想マシンでは，元の機械語コードにおけるレジスタやメモリを，別のレジスタやメモリでシミュレートしない．つまり変換済みコードでも，元の機械語コードとまったく同じレジスタやメモリを，まったく同じように利用する．そのため，基本的には実行位置を調整するだけで，モードの切替えを実現することができる．

仮想実行モードへ切り替える場合には，対象プログラムの現在の実行位置を仮想マシンの適切なデータ領域に設定する．そして仮想マシンに，4.1 節のステップ（i）から（iv）までの処理を実行させる．また通常実行モードへ切り替える場合には，現在実行中の変換済みコードに対応している元の機械語コードへ，実行位置を調整する．

提案環境のリセット 仮想実行モードにおいて，ハンドラの登録・抹消や逆実行の切替えにより設定が変更された場合には，提案環境を適切にリセットする必要がある．まず仮想マシンでは，キャッシュに保存されている変換済みコード片をすべて削除する．また Directing Library では，仮想マシンから読み込んだ拡張デバッグ情報（4.2.2 項）をすべて破棄する．そして変更された設定に関する情報を，仮想マシンの適切なデータ領域に設定する．最後に仮想実行モードで実行を再開させることにより，新しい設定が提案環境に反映される．

4.3 実装上の制約

本節では，提案環境の現在の実装に存在する主な制約について述べる．

4.3.1 システムコールの監視

現在，提案環境で監視することができるシステムコールは，メモリ操作やファイル操作に関するものなど，非常に頻繁に利用されるものに限られている．その他のシステムコールに関しては，提案環境は監視をまったく行わず，単純にそのまま実行する．そのためイベントの検出や逆実行などを行うことができない☆．

提案環境において，システムコールの監視に関わる部分は，現在も特に積極的に開発が続けられている部分である．以上の制限の解決に向けて，順次，監視可能なシステムコールを増やしていく計画である．

☆ 6 章の評価で利用した SPEC CPU2000 などのベンチマークで利用されたシステムコールは，すべて監視可能である．

4.3.2 シグナルハンドラの実行

提案環境は，対象プログラムの実行中にシグナルを検出すると，対象プログラムの実行を一時停止し，director 側に制御を戻す．この時点で，director は，対象プログラムの状態を調査することや，対象プログラムに対しシグナルハンドラを実行させることが可能である．ただし提案環境では，仮想実行モードで実行させることができるシグナルハンドラの種類に制限が存在する．

シグナルには，非同期シグナルと同期シグナルが存在する．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）．そのため仮想実行モード自体は，発生するシグナルの種類やタイミングに対して，特に影響を与えない．これに対し同期シグナルは，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）．そのため，仮想マシンで変換中にエラーが起こった場合など，本来とはまったく異なる種類とタイミングのシグナルが発生する場合がある☆．

このように，提案環境の仮想実行モードで発生する同期シグナルは，対象プログラムの実行中に本来発生するべきものとは異なる可能性がある．そのような場合に，不用意にハンドラを実行してしまうと，対象プログラムの実行に深刻な問題が発生する可能性が高い．そのため提案環境の仮想実行モードでは，同期シグナルに対するハンドラの実行を禁止している．そして非同期シグナルに対するハンドラに限り，その実行を許可している（その場合には，ハンドラは仮想マシン上で実行される）．

同期シグナルは，主にエラーの発生を示すものである．そのため，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．しかしそれ以外のアプリケーションを提案環境で実行する場合には，同期シグナルを発生するコードの周辺を通常実行モードで実行する，などの工夫を行う必要がある．

5. サンプル director：可逆デバッガ

本章では，提案環境を利用して実際に構築した director の例として，可逆デバッガを紹介する．可逆デバッガとは，従来のデバッガで提供されているような基本的な実行の制御の機能に加えて，**プログラムの逆実行**の機能を提供するデバッガのことである．

提案環境を利用すると，このような可逆デバッガも容易に構築可能である．3.2 節で述べたように，Directing Library は Directing API を通して，基本的な実行の制御に必要な機能（表 1 (b)）や，逆実行に必要な機能（表 1 (c)，(e)）を標準で提供している．そのため基本的には，ユーザインタフェースに関する部分を実装し，内部でそれぞれの機能に対応した Directing API を呼び出すだけで，文献 20) のような単純な可逆デバッガと同等のものを構築することができる．

さらに本研究では，Directing API を利用してデバッグに有用な様々な機能を実装し，このように構築した可逆デバッガの拡張を行っている．実装した機能としては，ウォッチポイント[19]，ヒープ検査[8]，動的スライシング[10] などがある．以下では，このような機能の中で，特にヒープ検査の機能について詳しく述べる．

5.1 ヒープ検査

メモリに関するエラーは，発見することが非常に困難なエラーである．これはエラーの影響が，エラーが発生した直後には現れず，まったく関係のない位置で現れることが多いためである[8]．そこで本研究では，Directing API を利用して，ヒープメモリに関するエラーを検出するための機能を実装した．

以下では，まず同様のエラーを検出することができるツールとして Dmalloc[6] と Purify[8] を取り上げ，その後，本研究で実装した機能について述べる．本論文では，それぞれの手法の違いを明確にするために，非常に一般的なエラーである境界を越えたアクセスや，解放済みのメモリに対するアクセスなどを検出するための機能に，特に焦点を当てて紹介する．

5.1.1 Dmalloc

Debug Malloc Library（Dmalloc）[6] は，システム標準のヒープ処理関数（e.g. malloc()）を独自のもので置き換え，置換したヒープ処理関数において様々な検査を行うツールである．たとえば境界を越えた書き込みや，解放済みのメモリに対する書き込みは以下のように検出する：

- メモリを割り当てる際に，要求されたサイズよりも大きい領域を割り当て，その両端の領域をダミーデータで初期化する．
- メモリを解放する際に，解放する領域をダミーデータで埋める．
- 適切なタイミングで（ライブラリが一定回数呼び出されたときなど），ダミーデータが別の値で上書きされていないかを検査する．

ただしこの手法には，エラーを発生と同時（ダミー

☆ たとえ仮想マシンのコードに問題がなくても，対象プログラムのエラーが，これらのコードにおいてまったく別の形で露呈することもある．

データの上書きと同時）に検出できない，読み込みに関するエラーは検出できない，といった問題点がある．

5.1.2 Purify

Purify[8] は，ヒープ処理関数だけではなく，すべてのメモリの読み書きも監視対象とすることにより，Dmalloc の問題点を解決している．これは，提案環境と同様に，対象プログラムやライブラリの機械語コードを変換し，メモリの読み書きの直前に監視コードを挿入することにより実現している．ただし提案環境とは異なり，Purify による変換は，**実行前**（実行ファイル作成時など）に行われる．たとえば境界を越えたアクセスや，解放済みのメモリに対するアクセスは以下のように検出する：

- 各メモリアドレスの割当て状態を表すビットテーブルを用意し，割当てや解放を行う際に更新していく．ただし割当てを行う際には，要求されたサイズよりも大きい領域を割り当て，その両端の領域を表すビットは未割当て状態にする．
- 個々のメモリの読み書きの直前にビットテーブルを調べ，その対象アドレスの割当て状態を検査する．

Purify の手法には，エラーを発生と同時（読み書きと同時）に検出できる，読み込みに関するエラーも検出できる，といった利点がある．また Purify は，未初期化メモリの利用など，他の様々な種類のエラーも検出することができる非常に高機能なツールである．しかし，対象プログラムやライブラリの変換を実行前に行う必要があり，柔軟性に欠けるといった問題もある．

5.1.3 本研究で実装したヒープ検査機能

上述したように，本研究で構築している可逆デバッガは，逆実行[3),4)]，ウォッチポイント[19)]，ヒープ検査[8)]，動的スライシング[10)] など，デバッグに有用な様々な機能を提供している．ただしこれらの機能は，いずれも相応のオーバヘッドをともなう機能である．そのためユーザが，対象プログラムの実行を制御しながら，必要に応じてこれらの機能を有効にできることが望ましい．このような要求は，Dmalloc や Purify のような独立したテストツールでは存在しなかったものである．

提案環境の大きな利点の 1 つとして，その柔軟性があげられる．Purify とは異なり，提案環境では対象プログラムやライブラリの変換を実行中に行う．そのため，対象プログラムの実行中であっても自由に変換内容を変更することが可能であり，上記のような要求も容易に実現することができる．

本研究で実装したヒープ検査機能では，Purify と同様に，ヒープ処理関数とすべてのメモリの読み書きの監視を行う☆．**表 3** に，本機能が利用するハンドラの一覧を示す．ヒープ処理関数の監視を行うハンドラは，表 2 の CALL_{rep} のイベントを利用して，システム標準のヒープ処理関数を置換するためのものである．また __check_range() は，表 2 の LOAD_{pre} と STORE_{pre} のイベントを利用して，すべてのメモリの読み書きの監視を行うためのハンドラである．

表 3 ヒープ検査で利用するハンドラ
Table 3 Handlers used in heap checking.

ハンドラ	イベント
ヒープ処理関数の監視	
__malloc_rep()	CALL_{rep}
__calloc_rep()	CALL_{rep}
__realloc_rep()	CALL_{rep}
__free_rep()	CALL_{rep}
メモリの読み書きの監視	
__check_range()	LOAD_{pre}，STORE_{pre}

Purify と同様に，ヒープ処理関数の監視を行うハンドラでは，割当て状態を表すビットテーブルを作成する．同時に，メモリリークや不正な領域の解放などの簡単なエラーの検出も行う．そして __check_range() において，読み書きの対象アドレスの割当て状態を検査し，境界を越えたアクセスや，解放済みのメモリに対するアクセスの検出を行う．

ただし，すべてのメモリの読み書きの監視を行うためには，非常に大きなオーバヘッドが必要となる．そこで本研究で実装したヒープ検査機能では，提案環境の柔軟性を利用して，次の 2 種類の監視を行う：(a) ヒープ処理関数のみを監視，(b) ヒープ処理関数とすべてのメモリの読み書きを監視．提案環境では，この (a) と (b) の監視の切替えを，対象プログラムの実行中に自由に行うことが可能である．そのためユーザは，たとえば通常は (a) の監視のみを行い，エラーがあると思われる実行区間のみ (b) の監視を行うことができる（もちろん，ヒープ検査機能を単独で利用するだけではなく，逆実行など他の機能と組み合わせて利用することも可能である）．

提案環境を利用することにより，このようなヒープ検査の機能も数百行のコードで簡単に実装することが可能となる．本研究では，ハンドラ自体のために約 360 行のコードを記述し，可逆デバッガに対し約 60 行のコードを追加することで実装を行った．ヒープ検査に必要なプログラムの実行の監視を行う機能は，本来は実装に大変な労力が必要となる機能である．しか

☆ ただし実験段階の機能なので，検出できるエラーの種類は Purify に比べ限られている．

し提案環境を利用することにより，実際の監視内容にあたる部分のみをハンドラとして記述するだけで，簡単にプログラムの実行の監視を実現することが可能となる．

6. 評　　価

本章では，提案環境の評価について述べる．評価では，提案環境を従来の環境と比較する際に，Dynascope[16] ではなく LVM[7] を利用した．これは，提案環境が動作する Intel x86 アーキテクチャ上では，Dynascope が動作しないためである．

評価は CPU に Pentium4 2.4 GHz を，メモリに 512 MB を搭載した計算機上で行った．使用したシステムの主なコンポーネントのバージョンは，以下のとおりである：Linux Kernel 2.4.27，gcc 2.95.4，glibc 2.3.2.

6.1 従来の環境との比較

2 章で述べたように，従来のプログラム実行制御・監視環境には，(1) 既存の開発環境との互換性と，(2) 実行速度の面で大きな問題がある．本節では，提案環境を LVM[7] と比較するために行った評価について述べる．

評価対象には，Stanford Integer Benchmark Suite を利用した．ただし LVM の評価の際には，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する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）．これに対し，提案環境の評価の際には，ベンチマークを既存の開発環境でコンパイルしたものをそのまま利用できた．このことから，提案環境では，従来の環境の（1）互換性の問題を解決していることが分かる．

表 4 に LVM と提案環境の性能の評価結果を示す．表中の数値は，ベンチマークを CPU 上で直接実行した場合の実行時間と，それぞれの環境で実行した場合の実行時間との比率を表している．“BASE” は，実行の監視をいっさい行わなかった場合の性能を表す．そのような場合には，提案環境では通常実行モードで実行を行う（図 1 (a)）．そのため，Directing API の呼び出しのコストは別途必要となるが，対象プログラムそのものの実行に関しては，まったくオーバヘッドが発生しない．また “STORE” は，メモリの書き込みのイベントに対し，空のハンドラを登録した場合の性能を表す．同様に “LOAD & STORE” は，メモリの読み書きのイベントに対し，空のハンドラを登録した場合の性能を表す．

表 4 LVM との比較
Table 4 Comparison with LVM.

	BASE	STORE	LOAD & STORE
LVM	78.5	231.2	773.6
提案環境	(1.0)	11.7	37.9

（CPU 上で直接実行した場合の実行時間との比率）

表 4 に示したように，実行の監視を行わなかった場合には，提案環境は LVM よりも 78.5 倍も高速な実行を実現している．またメモリの書き込みを監視した場合には 19.8 倍，メモリの読み書きを監視した場合には 20.4 倍も高速な実行を実現している．これらのことから，(2) 実行速度に関して，提案環境は従来の環境に比べ非常に高速であることが分かる．

6.2 ヒープ検査

本節では，提案環境を利用して実際に構築した director の性能の評価について述べる．評価対象には，SPEC CPU2000[18] ベンチマークから，整数演算に関するもの（CINT）8 個と，小数演算に関するもの（CFP）4 個を選択して利用した．個々のベンチマークは，トレーニングサイズの入力データを用いて実行した．

表 5 に，5.1 節で述べたヒープ検査の機能の性能の評価結果を示す．表中の数値は，ベンチマークを CPU 上で直接実行した場合の実行時間と，ヒープ検査の機能を利用しながら実行した場合の実行時間との比率を表している．表中の “監視（a）” と “監視（b）” は，本研究で実装したヒープ検査の機能の評価結果を示しており，それぞれ 5.1.3 項で述べた 2 種類の監視に対応している．

表 5 の数値の平均より，監視（a）の場合には CINT で平均 2.7 倍，CFP で平均 1.6 倍のオーバヘッドが生じた．同様に監視（b）の場合，CINT で平均 39.2 倍，CFP で平均 33.6 倍のオーバヘッドが生じた．6.1 節で述べたように，Stanford Integer Benchmark Suite を用いて評価した結果では，LVM では監視をまったく行わずに実行した場合でも，78.5 倍のオーバヘッドが生じた．また監視（b）のように，メモリの読み書きの監視を行った場合には，空のハンドラを登録した場合にさえ，773.6 倍のオーバヘッドが生じた．そのため，LVM を利用して同様のヒープ検査の機能を実装した場合には，提案環境を利用して実装したものよりも，はるかに低速になるものと予想される．

また表 5 には，Purify の性能の評価結果も示した．表 5 の数値の平均より，Purify では CINT で平均 55.2 倍，CFP で平均 57.9 倍のオーバヘッドが生じた．ただし Purify では，本研究で実装した機能よりも多く

表 5 ヒープ検査にともなうオーバヘッド
Table 5 Overhead of heap checking.

	ベンチマーク（CINT）								ベンチマーク（CFP）			
	gzip	vpr	mcf	crafty	parser	gap	vortex	bzip2	mesa	art	equake	ammp
監視（a）	1.9	1.9	1.5	2.1	4.3	3.8	4.8	1.2	1.8	1.2	1.9	1.4
監視（b）	33.2	36.2	12.5	39.2	42.5	61.5	54.5	34.0	19.4	15.0	70.7	29.4
Purify	57.0	47.8	16.2	46.0	63.6	67.9	73.3	69.8	120.3	20.1	58.0	33.3

（CPU 上で直接実行した場合の実行時間との比率）

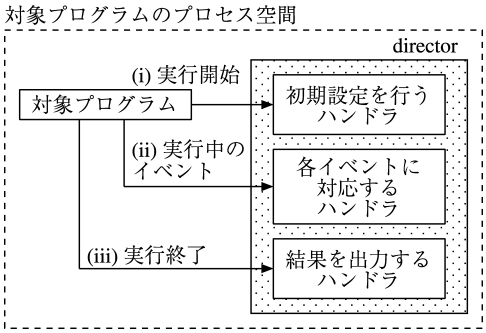


図 5 その他のフレームワークによる director の例
Fig. 5 Director developed by other frameworks.

の種類のエラーの検査を行う．たとえば未初期化メモリの利用は，検査に比較的大きなコストがかかるエラーである．そのため，本研究で実装した機能の性能と Purify の性能とを，単純に比較することはできない．しかし表 5 の結果より，提案環境を利用して構築した director の性能が，商用の製品に比べて極端に悪くなるということは，なさそうであることが分かる．

7. 関 連 研 究

2 章で述べたプログラム実行制御・監視環境以外にも，C 言語のプログラムを対象とした director の開発に利用できるフレームワークがいくつか提案されている．しかしその多くは，単一のプログラムの実行の監視のみを実現するものであった[9),11)~13),17)]．

これらのフレームワークでは，対象プログラムに登録されたイベントハンドラが，そのまま director として動作する．**図 5** にそのような director の例を示す：

(i) 最初に，対象プログラムの実行開始のイベントを捕捉し，director の初期設定を行う．
(ii) 実行中に発生するイベントを捕捉し，それぞれのイベントに対して必要な処理を行う．
(iii) 最後に，実行終了のイベントを捕捉し，監視結果を出力する．

これらのフレームワークでは，実行を制御するための機能が提供されていない．そのため開発できる director が，対象プログラムの実行を開始から終了まで監視し，最後に全体の監視結果を出力するようなものに限られるという問題がある．そのようにプログラム全体の監視を行うと，実行に非常に長い時間が必要となる場合も多い．また監視によって得られる情報も，プログラム全体を通した傾向のみに限られ，細かい情報を得ることはできない．

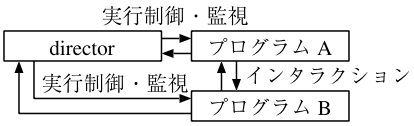


図 6 マルチプロセスに対応した director の例
Fig. 6 Director that can handle multi-processing.

これに対し本研究の提案環境では，プログラムの実行の監視を行うための機能だけではなく，実行の制御を行うための機能も提供されている．そのため提案環境では，5 章で述べた可逆デバッガのように，対象プログラムの実行の制御を行いながら，特に興味のある部分のみを詳細に監視することが可能である．

またこれらのフレームワークでは，director の対象プログラムは，単一のプログラムのみに限られるという問題がある．これは，図 5 に示したように，director が特定の対象プログラムのイベントハンドラとして実装されるためである．これに対し本研究の提案環境では，図 1 に示したように，director は対象プログラムとは独立したプログラムとして実装される．そのため提案環境では，マルチプロセスに対応した director を構築することも可能である（**図 6**）．

8. ま と め

本論文では，C 言語のプログラムを対象とした director の開発に利用できる，新しいプログラム実行制御・監視環境を提案した．プログラム実行制御・監視環境は，プログラムの実行時の振舞いを調査するための様々なツール（director）の構築に利用可能である．このようなツールは，プログラムの不具合の検出や，その原因の特定，開発者のプログラムに対する理解の促進などに欠かすことができないものである．

しかし従来の環境には，(1) 既存の開発環境との互

換性と，(2) 実行速度の面で大きな問題があった．提案環境では，機械語プログラムの実行の制御と監視を行うことにより，(1) の問題を解決している．また提案環境では，Dynamic Translation を行い，対象プログラムと監視コードの実行を機械語コードによって直接行うことにより，(2) の問題も大幅に改善している．

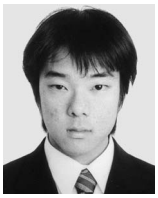
今後の課題としては，マルチスレッドプログラムへの対応があげられる．マルチスレッドプログラムの動作は，シングルスレッドプログラムの動作と比べて非常に複雑である．そこで本研究では，スレッドのスケジューリングのタイミングなどを完全に制御できる，ユーザレベルスレッドシステムを提案環境に導入し，マルチスレッドプログラムに対応した director の構築を支援できるように，提案環境を拡張することを検討中である．

謝辞 本研究は，文部科学省科学技術振興調整費「環境情報獲得のための高信頼性ソフトウェアに関する研究」の支援による．

参 考 文 献

1) Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A transparent dynamic optimization system, *Proc. ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation*, pp.1–12 (2000).
2) Ball, T. and Larus, J.R.: Optimally profiling and tracing programs, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1319–1360 (1994).
3) Chen, S., Fuchs, W.K. and Chung, J.: Reversible Debugging Using Program Instrumentation, *IEEE Trans. Softw. Eng.*, Vol.27, No.8, pp.715–727 (2001).
4) Cook, J.J.: Reverse Execution of Java Bytecode, *The Computer Journal*, Vol.45, No.6, pp.608–619 (2002).
5) Crescenzi, P., Demetrescu, C., Finocchi, I. and Petreschi, R.: Reversible Execution and Visualization of Programs with LEONARDO, *Journal of Visual Languages and Computing*, Vol.11, No.2, pp.125–150 (2000).
6) Debug Malloc Library. http://dmalloc.com
7) Demetrescu, C. and Finocchi, I.: A portable virtual machine for program debugging and directing, *Proc. 2004 ACM Symp. on Applied Computing*, pp.1524–1530 (2004).
8) Hastings, R. and Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors, *Proc. Winter USENIX Conf.*, pp.125–136 (1992).
9) Jeffery, C., Zhou, W., Templer, K. and Brazell, M.: A lightweight architecture for program execution monitoring, *Proc. 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.67–74 (1998).
10) Korel, B. and Yalamanchili, S.: Forward Computation of Dynamic Program Slices, *Proc. 1994 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pp.66–79 (1994).
11) Larus, J.R. and Schnarr, E.: EEL: Machine-Independent Executable Editing, *Proc. ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation*, pp.291–300 (1995).
12) Nethercote, N. and Seward, J.: Valgrind: A Program Supervision Framework, *Electronic Notes in Theoretical Computer Science*, Vol.89, No.2 (2003).
13) Ronsse, M., Maebe, J. and Bosschere, K.: Detecting Data Races in Sequential Programs with DIOTA, *Proc. 10th Int'l Euro-Par Conf.*, pp.82–89 (2004).
14) Rosenberg, J.B., 吉川邦夫 (訳)：デバッガの理論と実装，アスキー出版局 (1998).
15) Scott, K., Velusamy, N.S., Childers, B., Davidson, J. and Soffa, M.L.: Retargetable and reconfigurable software dynamic translation, *Proc. Int'l Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pp.36–47 (2003).
16) Sosic, R.: Design and Implementation of Dynascope, a Directing Platform for Compiled Programs, *Computing Systems*, Vol.8, No.2, pp.107–134 (1995).
17) Srivastava, A. and Wall, D.W.: ATOM: A System for Building Customized Program Analysis Tools, *Proc. ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation*, pp.196–205 (1994).
18) The Standard Performance Evaluation Corporation (SPEC). http://www.specbench.org
19) Wahbe, R., Lucco, S. and Graham, S.L.: Practical Data Breakpoints: Design and Implementation, *Proc. ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, pp.1–12 (1993).
20) 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).

(平成 17 年 3 月 31 日受付)
(平成 17 年 10 月 11 日採録)



孝壽 俊彦

2003 年慶應義塾大学理工学部卒業．2005 年同大学大学院理工学研究科修士課程修了．同年同博士課程入学，現在に至る．ソフトウェア工学に関する研究に従事．日本ソフトウェア科学会会員．

高田 眞吾（正会員）

1990 年慶應義塾大学理工学部卒業．1992 年同大学大学院理工学研究科修士課程修了．1995 年同博士課程修了．博士（工学）．同年奈良先端科学技術大学院大学情報科学研究科助手．1999 年より慶應義塾大学理工学部情報工学科専任講師．ソフトウェア工学，情報検索等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE CS 各会員．

土居 範久（正会員）

1969 年慶應義塾大学大学院博士課程単位取得退学．慶應義塾大学理工学部教授を経て，2003 年より中央大学理工学部教授，慶應義塾大学名誉教授．工学博士．現在，日本学術会議会員・第 3 部副部長，文部科学省科学技術・学術審議会委員，総務省情報通信審議会委員，科学技術振興機構（JST）社会技術研究開発センター「情報と社会」領域統括，特定非営利活動法人日本セキュリティ監査協会会長，国際計算機学会（ACM）日本支部長，等．専門はソフトウェアを中心とした計算機科学．情報処理学会名誉会員．