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

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，実際に開発中のプログラムを実行し，その振舞いを調査する作業が頻繁に繰り返される．これは，実際にプログラムを実行してみることが，不具合の検出や，その原因の特定，プログラムに対する理解などに欠かすことができないためである．このような作業を支援するツールを，総称して director と呼ぶ．Director は，その種類や目的にかかわらず，プログラムの実行の制御と監視の 2 種類の基本機能を必要とする場合が多い．そこで筆者らは以前の研究で，director 開発者に対しこれらの基本機能を提供する，プログラム実行制御・監視環境を提案した．しかし筆者らの環境には，マルチスレッドプログラムの実行を考慮していないという問題があった．そこで本論文では，マルチスレッドプログラムに対応した director の開発のための，プログラム実行制御・監視環境を提案する．提案環境では，マルチスレッドプログラムの実行の監視を非常に柔軟に行うことができる．Director はこの機能を利用することで，必要最小限の部分のみを監視し，オーバヘッドを抑えることなどが可能となる．提案環境は，ユーザ空間スレッド機構を組み込んだ仮想マシンを利用して，このような機能を実現している．また本論文では，実際に提案環境を利用して構築した director の例を紹介し，その評価結果についても述べる．

A Directing Platform for Multi-threaded Programs

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

Executing a program under development and examining its behavior is a frequently repeated task during software development. Such tasks are necessary for detecting bugs, finding the causes of bugs, understanding the program itself, and etc. Tools which can assist such tasks are called directors. In general, directors require two basic functionalities: control and monitor of program executions. Therefore, in our previous research, we proposed a directing platform to offer these functionalities to director developers. However, our directing platform did not consider execution of multi-threaded programs. In this paper, we focus specifically on directors for multi-threaded programs, and propose a new directing platform. Our directing platform enables flexible monitoring of multi-threaded program executions. This is important for directors since it enables their overhead to be limited only at the necessary parts of the executions. Our platform realizes such functionalities by incorporating a user space threading system into a virtual machine and executing multi-threaded programs on it. We developed an example director using our platform. In this paper, we also describe the details of this director and the evaluation results.

1. 序　　論

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を総称して director[6] と呼ぶ．Director はソフトウェア開発の様々な場面において利用されている．たとえばソフトウェアのテストの一環として，特定の種類の不具合を自動的に検出するエラーチェッカ[9] を利用できる；発見した不具合の原因を特定するためには，プログラムの実行を詳細に追跡するデバッガ[17] を利用できる；開発者がプログラムに対する理解を深めるためには，その挙動をグラフィカルに可視化するビジュアライザ[5] を利用できる；プログラムのボトルネックを発見するためには，その性能を計測するプロファイラ[3] を利用できる．

このように director の利用範囲は広く，目的に応じて様々な種類のものが存在する．しかし director は，共通して，プログラムの実行の制御と監視の 2 種類の基本機能を必要とする場合が多い．

プログラムの実行の制御　プログラムの実行を管理するための機能や，振舞いを操作するための機能．たとえば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を行う機能である．

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

プログラムの実行の監視 プログラムが実行時に示す様々な挙動（メモリの読み書き，スレッドの生成や終了など）は，**イベント**と見なすことが可能である．このようなイベントを捕捉し，関連した情報の収集を行う機能である．

プログラムの実行の制御と監視の機能の実装には，比較的大きな労力が必要となる．そこで筆者らは以前の研究で，これらの基本機能を提供するプログラム実行制御・監視環境を提案した[26]．Director 開発者は，これを利用することにより，director 固有の部分の開発に集中し，開発コストを抑えることが可能になる．

しかし筆者らのプログラム実行制御・監視環境には，マルチスレッドプログラムの実行を考慮していないという問題があった．マルチスレッドプログラムの動作は非常に複雑であり，その実行時の振舞いの調査には，director が非常に有用である．そこで本論文では，マルチスレッドプログラムに対応した director の開発のための，プログラム実行制御・監視環境を提案する．

以下，まず 2 章で，プログラム実行制御・監視環境について紹介する．そして 3 章で，マルチスレッドに対応したプログラム実行制御・監視環境を提案する．4 章で，提案環境の実装について述べる．5 章で，実際に提案環境を利用して構築した director の例を紹介する．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

筆者らは，従来よりプログラム実行制御・監視環境の研究をしており，特にそれまでの環境[6),20)]にあった互換性と実行速度の問題に着目していた[26]．本章では，その概要について紹介する．

本環境は，Intel x86 アーキテクチャの Linux 上で動作し，C 言語で記述された一般的なユーザプログラムを，director の調査対象とする．**図 1** に，本環境を利用する director の概観を示す．本環境は，仮想マシンと Directing Library の 2 つのコンポーネントからなる：

仮想マシン 実行の監視を行うために，調査対象プログラムを仮想マシン上で実行する．仮想マシンは，調査対象プログラムの実行を続けながら，director の興味のある**イベント**を見張る．仮想マシンはイベントを捕捉すると，director によって登録された**ハンドラ**を呼び出す．Director 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．

Directing Library Director に対し Directing API を提供する役割を持つ．Director はこの Directing API を利用して，調査対象プログラムの実行の制御と監視を行う．Directing Library は，内部で仮想マシンの操作を行う．しかしその詳細は，可能な限り Directing Library 内で隠蔽される．そのため director 側は，基本的に，仮想マシンの操作の詳細について意識する必要はない．

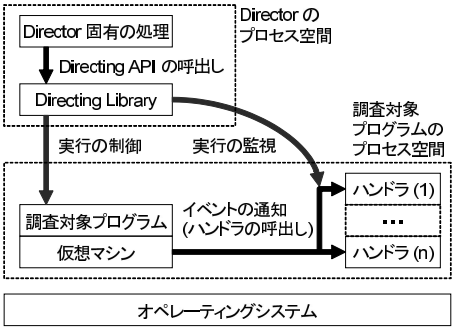


図 1 プログラム実行制御・監視環境[26)]
Fig. 1 Overview of the directing platform [26)].

仮想マシンは，調査対象プログラムを既存の開発環境でコンパイルし，生成された機械語コードをそのまま実行する．これにより，従来の環境[6)]にあった互換性の問題を解決している☆．また仮想マシンは，内部で Dynamic Translation[2)] 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技術であり，Java の JIT コンパイラ[1)] などでも利用されている技術である．これにより，従来の環境[6),20)]にあった実行速度の問題を大幅に改善している．

しかし本環境では，マルチスレッドプログラムの実行を考慮していないという問題が残されたままであった．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．このような複雑な振舞いの調査には，director が非常に有用である．

3. 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

本論文では，マルチスレッドプログラムに対応した

☆ 仮想マシンは，Intel i386，および i486 互換の一般的な命令を解釈・実行できる．また Directing Library では，実行の制御と監視を高級言語レベルで行うための機能も提供している．

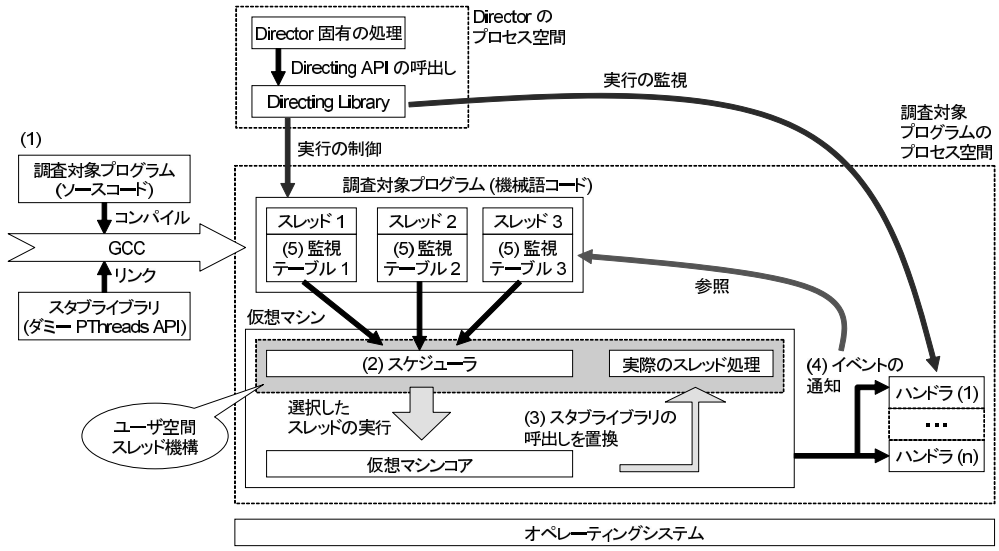


図 2 提案環境の概要
Fig. 2 Overview of our proposed platform.

プログラム実行制御・監視環境を提案する☆．提案環境は，2 章で述べた環境を基盤として設計を行った．提案環境の主な設計方針は，以下のとおりである：

互換性 Director を利用するために，調査対象プログラムのソースコードを大きく変更する必要があるのは望ましくない．そこで提案環境は，多くのシステムで利用されている，POSIX Threads（Pthreads）API[23] とできるだけ互換性を持たせることにした．

スレッド機構の制御 プログラム実行制御・監視環境は，調査対象プログラムの利用するスレッド機構を完全に制御下におく必要がある．そこで提案環境では，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[15] を利用することにした．

柔軟なスレッドの実行の監視 プログラムの実行の監視は，オーバヘッドが非常に大きい機能である．そのため必要な部分のみ監視を行えるような柔軟性が望まれる．そこで提案環境では，スレッドごとに監視内容を細かく設定できるようにすることにした．

以下では，提案環境の概要と，director 開発者から見た提案環境の主な機能について詳しく紹介する．

3.1 提案環境の概要

図 2 に提案環境の概要を示す．提案環境でも，調査対象プログラムをコンパイルして生成された機械語コードを，仮想マシン上で実行する（図 2 (1)）．提案環境では，調査対象プログラムが利用するスレッド API として，Pthreads API[23] を前提にしている．そして調査対象プログラムは，コンパイル時に，提案環境が提供する**スタブライブラリ**とリンクする必要がある．このスタブライブラリには，ダミーの Pthreads API が含まれている．

ここで，調査対象プログラムとスタブライブラリをリンクするためには，調査対象プログラムの Makefile などに多少の変更が必要となる．調査対象プログラムの構築方法はそれぞれ異なるため，この変更は director の利用者が自らの手で行う必要がある．しかし Pthreads API 自体には基本的に互換性があるので，調査対象プログラム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．

本研究では，提案環境の仮想マシンに対し，ユーザ空間スレッド機構[15] を組み込んだ．ユーザ空間スレッド機構とは，スレッドの存在を OS がいっさい関知せず，純粋にユーザ空間内で実装されたスレッド機構である．そのため提案環境にとって，制御することが容易であり非常に都合がよい．

仮想マシンのユーザ空間スレッド機構は，独自のスレッドスケジューラを持つ（図 2 (2)）．スケジューラは，次に実行するスレッドを選択し，それを仮想マシンのコアに実行させる．そして仮想マシンのコアは，スタブライブラリ内のダミーの Pthreads API の呼び出しを検出すると，それをユーザ空間スレッド機構へ

☆ 正式名称は Scopion である．これは Scope と Scorpion をかけたものである．

表 1 基本イベント
Table 1 Basic events in our platform.

イベント	説明
LINE，PROC	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

表 2 スレッドに関連したイベント
Table 2 Thread related events in our platform.

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
$\mathrm{SWITCH}_{pre,post}$	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
$\mathrm{LOCK}_{pre,post}$	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
$\mathrm{RDLOCK}_{pre,post}$	読み込み用のロックの獲得
$\mathrm{WRLOCK}_{pre,post}$	書き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
$\mathrm{WAIT}_{pre,post}$	待機状態に入る
BROADCAST	シグナルのブロードキャスト
SIGNAL	シグナルの送信

の呼び出しに置き換える（図 2 (3)）．そして実際のスレッドに関連した処理は，ユーザ空間スレッド機構内で行う．

提案環境でも，仮想マシンが，調査対象プログラムのイベントの捕捉と，対応する director のハンドラの呼び出しを行う（図 2 (4)）．提案環境の仮想マシンでは，**表 1** に示した基本的なイベントに加え，**表 2** に示したスレッドに関連した様々なイベントを監視可能である．表 2 のイベントの多くは，Pthreads API のスレッドモデル[23] をそのまま反映したものである．

提案環境では，監視内容を指定するための**監視テーブル**を導入している（図 2 (5)）．監視テーブルには，仮想マシンが捕捉するべきイベントと，呼び出すべきハンドラの組合せに関する情報が含まれている．監視テーブルは，各スレッドごとに設定することが可能である．また各スレッドが実行の途中であっても，変更することが可能である．そのため提案環境では，必要なスレッドの必要な実行区間のみ監視を行うことができ，非常に柔軟な監視が可能となっている．

また提案環境の Directing Library は，マルチスレッドプログラムの実行の制御と監視を行うための，様々な Directing API を提供している（**表 3**）．Directing Library は，内部で仮想マシンとユーザ空間スレッド機構の操作を行うことで，これらの機能を実現している．しかし提案環境では，その詳細を可能な限り Directing Library 内で隠蔽するように注意した．

3.2 提案環境の主な機能

本節では，Directing Library が director 開発者に対して提供する，Directing API について紹介する．表 3 に，Directing API の主な機能を示す：

(a) 提案環境下で，調査対象プログラムを起動・終了させるための機能である．提案環境では，“アタッチ” や “デタッチ” を行うための機能は提供していない．“アタッチ” とは，すでに提案環境外で実行中のプログラムに接続し，提案環境下で実行の制御と監視を行えるようにするための機能である．逆に “デタッチ” は，提案環境下で実行中のプログラムを，提案環境外に切り離すための機能である．

(b) 調査対象プログラムの実行の進捗を管理するための機能である．“プロセスの継続実行” では，プロセス中のすべてのスレッドが実行される．これに対して，“カレントスレッドの継続実行” や “カレントスレッドのステップ実行” では，カレントスレッドのみが実行される．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) 調査対象プログラムのスレッドや同期オブジェクトに関する情報を取得するための機能である．同期オブジェクトとは，スレッド間で同期をとるために利用されるオブジェクトであり，mutex，reader/writer lock，condition variable の 3 種類が利用可能である．

(d) 調査対象プログラムのメモリやレジスタの内容を調査するための機能である．“変数に関する情報の取得” と “メモリの読み書き” を組み合わせることで，ソースコードにおける変数の値などを調査することも可能である．

(e) スレッドの監視内容を指定するための機能である．“ハンドラのロードとアンロード” では，director のハンドラを含むモジュールを，調査対象プログラムのプロセス空間にロード・アンロードすることができる．Director 開発者は，このモジュールを，実行時にロード可能な動的ライブラリとして作成する．

表 3 Directing API の主な機能
Table 3 Main functionalities of Directing API.

実行の制御に関する機能		
(a)	起動と終了	調査対象プログラムを起動，または終了する．
(b)	プロセスの継続実行	プロセスの実行を再開する．
	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．
	カレントスレッドの継続実行	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，機械語コードの 1 命令 or ソースコードの 1 行分だけ実行する．
	ブレークポイントの設定	機械語コードの命令 or ソースコードの行にブレークポイントを設定する．
(c)	スレッドに関する情報の取得	プロセス中に存在するスレッドに関する情報を取得する．
	同期オブジェクトに関する情報の取得	プロセス中に存在する同期オブジェクトに関する情報を取得する．
(d)	変数に関する情報の取得	ソースコードにおける変数のメモリアドレス，型情報，スコープ情報などを取得する．
	メモリの読み書き	メモリの内容の取得や修正を行う．
	カレントスレッドのレジスタの読み書き	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．
実行の監視に関する機能		
(e)	ハンドラのロードとアンロード	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．
	監視テーブルの設定	スレッドの監視テーブルを設定する．

“監視テーブルの設定” では，仮想マシンが監視するべきイベントと呼び出すべきハンドラの組合せを，スレッドに設定することができる．表 1 と表 2 に，提案環境で監視可能なイベントの一覧を示す．さらに提案環境では，director が独自のイベント（カスタムイベント）を定義する機能も提供している．カスタムイベントに対しては，他の組み込みイベントとまったく同様に，ハンドラを登録することが可能である．5.2 節では，カスタムイベントを利用する director の例を取り上げる．

4. 提案環境の実装

本研究では，提案環境にユーザ空間スレッド機構[15]を組み込んだ．本研究では，そのために GNU Portable Threads (Pth)[7] ライブラリを利用した．Pth は，オープンソースのユーザ空間スレッドライブラリである．Pth を選択した理由は，以下のとおりである：(1) ソースコードが比較的小規模で，また非常にシンプルである，(2) Pthreads API のエミュレーションレイヤが存在する．

しかし提案環境に Pth を組み込むことは，単純な作業ではなかった．これは第 1 に，仮想マシンのコアから，単純に Pth の API を呼び出すことができなかったためである．実際には，仮想マシンのコア，Pth の API，Pth のスケジューラ間の遷移を考慮する必要があった．また第 2 の理由として，そのように Pth を組み込んだ仮想マシンを，別のプロセス空間に存在する Directing Library 側から操作する手法も，自明ではなかったことがあげられる．

4.1 仮想マシン

図 3 に，仮想マシンの概観を示す．本研究では，Pth

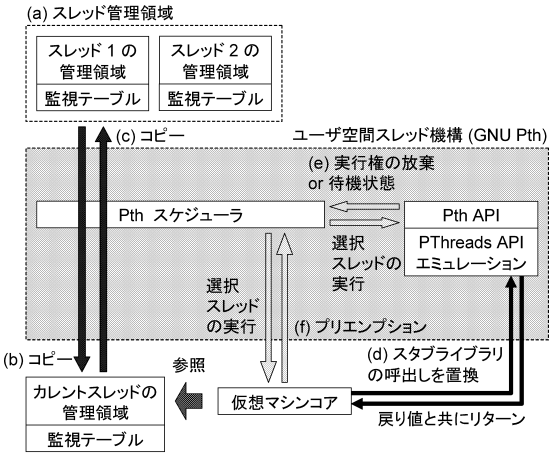


図 3 仮想マシンの概観
Fig. 3 Overview of our VM.

を仮想マシンに導入するにあたって，まず仮想マシンのデータ構造の拡張を行った．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監視テーブルなどは，スレッドごとに用意する必要がある．そこでそのようなデータ領域を，スレッド管理領域として分離した（図 3 (a)）．そして Pth のスレッド作成関数に修正を行った．まず新規スレッド作成時に，スレッド管理領域を割り当て，必要な初期化を行うようにした．また新規スレッドが，初めてスケジュールされたときに，自動的に仮想マシンのコアが呼び出されるようにした．

次に Pth のスケジューラを修正した．スケジューラは，仮想マシンのコアに，選択したスレッドの実行を再開させる必要がある．そこでスケジューラから各スレッドに切り替わる際に，そのスレッド管理領域を，仮想マシンのコアが参照するデータ領域にコピーするようにした（図 3 (b)）．仮想マシンのコアは，コピー

されたデータを利用して，各スレッドの実行を行う．また各スレッドからスケジューラに戻ってきた際には，仮想マシンのコアが参照したデータを，再びスレッド管理領域にコピーするようにした（図 3 (c)）．

そして仮想マシンのコアを修正し，スタブライブラリの呼び出しを，Pth の Pthreads API エミュレーションレイヤの呼び出しに置換する機構を追加した（図 3 (d)）．また Pth を修正し，スレッドに関連したイベント（表 2）を検出し，もし登録されていれば director のハンドラを呼び出す機構を追加した．さらに Pth のデータ構造を拡張し，スレッドや同期オブジェクトに関する情報を，Directing Library が解析しやすいようにした．

さらに本研究では，スレッドのプリエンプションも実現した．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 の関数を呼び出して待機状態に入った場合のみ，スレッドの切替えが行われる（図 3 (e)）．そこで本研究では，仮想マシンのコアを修正し，各スレッドのコードをある程度実行したら，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 (f)）．提案環境では，ソースコードにおける行数を用いて，仮想マシンのコアが各スレッドを一度に実行する量を指定できる．

4.2 Directing Library

Directing Library は，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，調査対象プログラムや仮想マシンの操作を行う．

4.2.1 実行の制御に関する機能

実行の制御に関する機能は，以下の 4 種類である：

起動と終了（表 3 (a)） “起動と終了” では，調査対象プログラムの起動と終了の処理を行うとともに，仮想マシンのロードとアンロードも行う．ここで前述したとおり，調査対象プログラムは，あらかじめスタブライブラリとリンクしておく必要がある．しかしスタブライブラリとリンクされたプログラムは，仮想マシンのユーザ空間スレッド機構がなければ実行できず，つねに提案環境下で実行されることになる．そのため提案環境では，“アタッチ” と “デタッチ” の機能は提供されていない．

実行の進捗の管理（表 3 (b)） これらの機能では，図 3 に示した仮想マシンのコア，Pth の API，Pth のスケジューラ間の遷移を操作する必要がある．ここで重要なことは，スレッドの切替えは必ずスケジューラを経由して行われることである．またスレッドの切替えは，各スレッドが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 (e)）と，プリエンプションが行われた場合（図 3 (f)）の 2 種類のタイミングで起こる．

“プロセスの継続実行” では，ユーザ空間スレッド機構に対する操作は行わず，プロセス全体の実行を再開する．“カレントスレッドの切替え” では，まず切替え先のスレッドの識別子を，スケジューラに設定する．そして強制的に，スケジューラに実行を切り替える．最後に，スケジューラに切替え先のスレッドを選択させ，そのスレッドの再開位置で停止させる．“カレントスレッドの継続実行” と “カレントスレッドのステップ実行” では，まず仮想マシンのコアを操作し，プリエンプション（図 3 (f)）を禁止させてから，継続実行やステップ実行の処理を行う．ただしプリエンプションを禁止していても，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 (e)）には，スレッドの切替えが起こる．その場合には，スケジューラを操作し，次のスレッドの再開位置で停止させる．“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で，容易に実現可能である．

スレッドや同期オブジェクトの情報の取得（表 3 (c)） “スレッドに関する情報の取得” や “同期オブジェクトに関する情報の取得” では，ユーザ空間スレッド機構内のデータを取り出し，解析を行う．これには，表 3 (d) の機能を活用している．そして，プロセス中に存在するスレッドや同期オブジェクトの一覧，個々のスレッドの情報などを取得する．

メモリやレジスタの調査（表 3 (d)） “変数に関する情報の取得” では，コンパイラが出力したデバッグ情報を基にして，必要な情報を取得する．“メモリの読み書き” では，単純に指定されたメモリアドレスの読み書きを行う．“カレントスレッドのレジスタの読み書き” では，レジスタを直接読み書きするだけではなく，レジスタが退避されている場合には，その退避領域に対して読み書きを行う．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[17] などが必要な場合には，表 3 (c) の機能を利用できる．

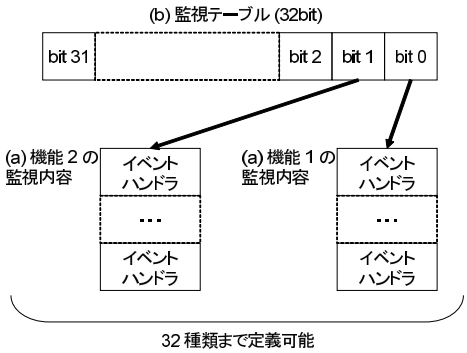


図 4 監視テーブルのデータ構造
Fig. 4 Data structure of a monitoring table.

4.2.2 実行の監視に関する機能

仮想マシンのモジュールには，動的ライブラリを操作するための関数が組み込まれている．“ハンドラのロードとアンロード” では，調査対象プログラムを操作して，これらの関数を呼び出させることにより，調査対象プログラム自身にロードとアンロードの処理を行わせる．またそれらの操作を行った際には，対応するデバッグ情報の読み込みや破棄なども行う．

図 4 に監視テーブルのデータ構造を示す．提案環境では，director は，director の機能ごとに必要な監視内容（イベントとハンドラの組合せの集合）を，あらかじめ定義しておかなければならない（図 4 (a)）．提案環境では，このような機能を，32 種類まで定義できる．監視テーブルは，実際には 32 bit のフラグであり，各ビットはそれぞれの機能に対応している（図 4 (b)）．そして監視テーブルのビットがオンであれば，そのビットに対応した機能の監視が行われることを表している．このようなデータ構造により，各スレッドに対する director の機能の有効化/無効化などを，単純なフラグの操作で行うことができる．

提案環境では，スレッドの監視テーブルを設定する方法は 2 種類ある．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機能である．Pthreads API では，各スレッドに対して様々な属性を設定することができる[15]．提案環境では，その拡張属性としてスレッドグループを追加した．そして Directing Library には，同一のスレッドグループに属するスレッドに対して，まとめて監視テーブルを設定する機能を追加した．

この機能を有効に活用するためには，調査対象プログラムのソースコードを修正し，スレッド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，提案環境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，必要な場合だけ埋め込むことができる．

4.3 制　　限

本節では，提案環境が現在持っている主な制限について述べる．

4.3.1 実装上の制限

提案環境では，Pthreads API[23] の仕様との完全な互換性は実現していない．たとえば提案環境では，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（スケジュールのポリシを設定するための属性や，mutex を異なるプロセス間で共有できるようにするための属性など）．これは，提案環境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，

また，以前の環境[26] に存在したプログラムの逆実行の機能は，提案環境では利用することができない．プログラムの逆実行[4] 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．しかし提案環境で逆実行を行うためには，スレッド間での同期など，解決しなければならない課題が多数残っている．

4.3.2 解析能力に関する制限

提案環境を利用して構築した director では，調査対象プログラムの正当性を保証することはできない．5 章で取り上げる data race 検出機能を持つデバッガのように，提案環境を利用して，プログラムのテストやデバッグの作業を支援するツールを構築することは可能である．しかし，そもそもテストやデバッグの作業だけでは，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[27]．

また提案環境には，マルチスレッドプログラムの実行の再現性に起因する制限が存在する．マルチスレッドプログラムの実行は，非決定的である．すなわち，マルチスレッドプログラムの実行時の振舞いは，コンテキストスイッチのタイミングに依存し，実行するたびに変わってしまう可能性がある．そのため提案環境を利用して，マルチスレッドプログラムの実行時の振舞いを解析する際には，解析したい特定の実行を提案環境下で再現する必要がある．いいかえれば，提案環境を利用して構築された director を最も有効に活用で

きるのは，そのような実行を提案環境下で容易に再現できる場合である．

しかしここで，提案環境には，コンテキストスイッチをソースコードにおける行を基準として行っているという問題がある．そのため提案環境では，それ以外の場所でコンテキストスイッチが発生する実行を再現することができない．これは，たとえば，そのようなコンテキストスイッチが発生する実行でのみ，不具合を生じるプログラムのデバッグを行う場合などに非常に大きな問題となる．そこで提案環境の次の版では，コンテキストスイッチをソースコードの行よりも細かい基準で行えるようにし，この制限を緩和する予定である．

また提案環境には，マルチスレッドプログラムの実行の記録・再生[18]を行う機能がないという問題がある．実行の記録・再生とは，非決定的なプログラムの実行を記録し，まったく同じ実行を再生するための機能である．そのため提案環境では，同じ実行を繰り返し行い，その振舞いを少しずつ調査して行くような場合などに不便が生じる．そこで現在，提案環境において実行の記録・再生の機能を実現する方法を検討しているところである．提案環境では，すでに調査対象プログラムの実行を監視する機能が実現されている．これを拡張し，調査対象プログラムの非決定性を生み出す振舞いを記録・再生する機構を追加する方法などが考えられる．

5. サンプル director

本研究では，実際に提案環境を利用して，data race 検出機能[19]を持つデバッガを構築した．Data race とは，スレッドが適切な排他制御を行わずに，共有変数にアクセスすることによって起こる，非常に一般的なエラーである．Data race を動的に検出する手法としては，Eraser[19]の Lockset Algorithm が有名である．Visual Threads[8]や Helgrind[14]などでも，これを少し拡張したアルゴリズムが利用されている．本研究では，これらを参考にして実装を行った．

以下では，まず基本となる Lockset Algorithm を紹介し，それから本研究での実装について述べる．

5.1 Lockset Algorithm

図 5 に，Lockset Algorithm の最も単純なバージョンを示す．Lockset Algorithm では，各共有変数 v に対して，v を保護していると考えられるロックの集合 $C(v)$ を関連付ける（lockset）．$C(v)$ の初期集合は，すべてのロックの集合である（図 5 (1)）．そして，スレッド t によって変数 v がアクセスされるたびに，スレッド t が獲得しているロックの集合と $C(v)$ の積集合をとり，$C(v)$ を更新する（図 5 (2-a)）．その結果，$C(v)$ が空集合になれば，v を保護しているロックは 1 つもないことになり，data race が起こる可能性があると警告を発生する（図 5 (2-b)）．

Let $locks\_held(t)$ be the set of locks held by thread t.
(1) For each v, initialize $C(v)$ to the set of all locks.
(2) On each access to v by thread t,
  (a) set $C(v) := C(v) \cap locks\_held(t)$;
  (b) if $C(v) = \{\}$, then issue a warning.

図 5 最も単純な Lockset Algorithm（文献 19) より引用）
Fig. 5 The simplest Lockset Algorithm (quote from Ref. 19)).

5.2 本研究における実装

本研究では，ヒープメモリ上で発生する data race の検出を行う機能を実装した．その際，排他制御を行うためのロック機構として，mutex のみを考慮した．

Eraser[19]では，図 5 のアルゴリズムを拡張し，メモリの初期化や読み込み専用メモリなどに関する問題も解決している．また Visual Threads[8]では，Thread Segment を導入して，そのアルゴリズムをさらに拡張している．本研究でも，それらの拡張部分の実装を行ったが，ここでは簡単のため，図 5 のアルゴリズムの実装部分についてのみ紹介する．

本研究の実装で利用している主なイベントハンドラは，次の 3 種類に分類できる：

ヒープ処理に関するハンドラ これらは，システム標準のヒープ処理関数（malloc() 関数など）と置換され，代わりに呼び出されるハンドラである．この置換には，表 1 の CALL_{rep} イベントを利用している．これらのハンドラでは，元のヒープ処理関数と同様に，ヒープメモリの割当てや解放を行う．また同時に，それらの操作を行ったことを通知するために，3.2 節（e）で述べたカスタムイベントを発生させる．

Mutex に関するハンドラ これらは，各スレッドのロックの獲得状況を追跡するためのハンドラである．表 2 の mutex に関連したイベントを監視し，ロックの獲得や解放が行われるたびに，それを行ったスレッドの獲得済みロック集合に関する情報を更新する．

Lockset に関するハンドラ これらは，実際に data race の検出を行うためのハンドラである．まずヒープ処理に関するハンドラによって発生させられるカスタムイベントの監視を行う．そして割り当

```
/* ハンドラを含むモジュールをロード */
1: module = scp_load_hook(scp, path);
/* 監視内容を定義するエントリを初期化 */
2: scp_init_hookset(scp, RACE_CHECK);
/* STORE_pre の監視を行うハンドラを登録 */
3: proc = scp_find_module_proc(
       module, "__sm_store");
4: scp_add_hook(
       scp, RACE_CHECK, SCP_STORE_PRE,
       proc, heap_start, heap_size, NULL);
```

図 6 ハンドラの登録を行うコードの例
Fig. 6 Example code for registering handlers.

てられたヒープ領域の lockset の初期化（図 5 (1)）などを行う．

また表 1 の LOAD_{pre} と STORE_{pre} のイベントの監視を行う．そして読書きが行われたヒープ領域の lockset を，スレッドの獲得済みロック集合との積集合に更新する（図 5 (2-a)）．さらに lockset が空集合になった場合には，警告を出力して実行を一時停止する（図 5 (2-b)）．

ハンドラを含むモジュールは，実行時にロード可能な動的ライブラリとして作成される．デバッガは，まずこのモジュールを調査対象プログラムのプロセス空間にロードする．そして 4.2.2 項で述べたように，機能ごとに必要な監視内容（図 4 (a)）をあらかじめ定義しておく．

例として**図 6** に，STORE_{pre} イベントの監視を行うハンドラを登録するまでのコードを，非常に簡単にしたものを示す．まず 1 行目では，パス名 *path* で示されるモジュールを，調査対象プログラムのプロセス空間にロードする．ここで *scp* は，調査対象プログラムを識別するために用いられるハンドルである．次に 2 行目では，RACE_CHECK という機能の監視内容を定義する準備として，そのエントリの初期化を行う．ここで RACE_CHECK は，監視テーブルのビットに対応する 0-31 の間の整数値である．そして 3 行目では，1 行目でロードしたモジュールから，lockset に関するハンドラである__sm_store() という名前の関数を検索する．最後に 4 行目では，RACE_CHECK の機能の監視内容に，STORE_{pre} イベントの監視を行うハンドラとして，検索した関数を登録する．*heap_start* と *heap_size* は，監視範囲を指定するためのオプションであり，ここではヒープメモリ領域を指定している．

本研究では，不必要な監視をできる限り避けるために，2 段階の監視を行うことにした．まずヒープ処理と mutex に関するハンドラを，監視テーブルに設定する機能の 1 つとして定義した．また lockset に関するハンドラは，別の機能として定義した．ヒープ処理と mutex に関するハンドラは，すべてのスレッドに対して，つねに有効にしておくものとする．それに対し，lockset に関するハンドラは，実行中に有効化/無効化を切り替えられるようにした．これにより，本機能を組み込んだデバッガの利用者は，調査対象プログラムの実行を制御しながら，特にエラーがあると思われるスレッドの，特定の実行区間のみに対し，data race の検出を行うことが可能となった．6 章では，これが director のオーバヘッドに与える影響について考察する．

```
1: thread = scp_get_threads(scp, SCP_ACTIVE);
2: scp_set_hookset(
       scp, thread, SCP_ADD_HOOKSET,
       (1 << RACE_CHECK));
3: scp_update_execution(scp);
```

図 7 ハンドラを有効化するコードの例
Fig. 7 Example code for enabling handlers.

図 7 に，RACE_CHECK の機能に登録したハンドラを有効にするコードの例を示す．まず 1 行目では，調査対象プログラムのアクティブなスレッドの一覧を取得する．次に 2 行目では，これらのスレッドの監視テーブルに対し，RACE_CHECK の機能にあたるビットをセットにする．最後に 3 行目では，監視テーブルの変更を反映させるために，調査対象プログラムの実行状態を更新する．

6. 評　　価

本章では，提案環境の評価について述べる．評価には，SPLASH-2（Stanford Parallel Applications for Shared Memory）[25] に含まれる 4 種類のカーネルベンチマークを用いた．**表 4** に，ベンチマークと指定したオプションの一覧を示す．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（inputs/tk29.O, -m22, -n2048, -n16777216），32 個のスレッドを使って問題を解くこと（-p32），さらに結果のセルフテストを行うこと（-t）を指定している．評価を行ったシステムの構成は，以下のとおりである：CPU Pentium4 2.4 GHz，Memory 512 MB，Linux Kernel 2.4.27, GCC 2.95.4，GLIBC 2.3.2.

6.1 仮想マシン

表 5 に，監視をいっさい行っていないときの，仮想マシンの実行速度の評価結果を示す．表中の “CPU” は，各ベンチマークを CPU 上で直接実行した場合の実行時間を表している．同様に “VM” は，各ベンチマークを仮想マシン上で実行した場合の実行時間を表

表 6 Data race 検出機能にともなうオーバヘッド（%）
Table 6 Overheads of our data race detector (%).

	Base	2 スレッド	4 スレッド	8 スレッド	16 スレッド	32 スレッド
cholesky	33.7	1137.3	1255.8	1530.7	2063.8	3269.7
fft	156.5	747.1	994.9	1519.2	2607.2	4886.5
lu	445.4	1452.6	2436.6	4357.8	8213.2	15461.1
radix	105.5	268.6	365.3	556.8	933.8	1692.9
平均	149.0	780.0	1050.5	1567.5	2563.7	4553.1

表 4 ベンチマーク
Table 4 Benchmarks.

ベンチマーク名	オプション
cholesky	inputs/tk29.O -p32 -t
fft	-m22 -p32 -t
lu	-n2048 -p32 -t
radix	-n16777216 -p32 -t

表 5 仮想マシンの実行速度
Table 5 Execution speeds of our virtual machine.

	cholesky	fft	lu	radix
CPU（sec）	5.9	16.0	30.0	20.9
VM（sec）	6.5	28.3	79.9	29.3
オーバヘッド（%）	10.0	77.2	166.8	40.2

している．そして “オーバヘッド” は，CPU 上で直接実行した場合に対し，仮想マシン上で実行した場合に発生したオーバヘッドの割合を表している．

表 5 より，提案環境の仮想マシンでは，10.0%から 166.8%（平均☆64.3%）のオーバヘッドが生じたことが分かる．これらの数字自体は，けっして小さなものではない．しかし次節で示すように，director の監視によっては，これらよりもはるかに大きいオーバヘッドが生じる．そのような場合には，提案環境の仮想マシン自体の実行速度が，大きな問題になることはないと思われる．

6.2 Data race 検出機能

本節では，5 章で紹介した data race 検出機能を持つデバッガの評価について述べる．**表 6** に，data race 検出機能にともなう，実行時間のオーバヘッドを示す．表中の数字は，CPU 上で直接実行した場合に対し，仮想マシン上でそれぞれの監視を行って実行した場合に発生したオーバヘッドの割合を表している．“Base” は，5.2 節で述べたヒープ処理と mutex に関するハンドラのみを有効にした場合を表している．また “2 スレッド” から “32 スレッド” までは，それらに加えて lockset に関するハンドラも有効にし，実際に data race の検出を行う場合を表している．上述したとおり，評価に用いたベンチマークでは，それぞれの問題を 32 個のスレッドを使って解く．そこで評価では，lockset に関するハンドラを有効にするスレッドの数を，2 から 32 まで増やしながら計測した．

表 6 から，ヒープ処理と mutex に関するハンドラのみを有効にした場合には，33.7%から 445.4%（平均 149.0%）のオーバヘッドが生じることが分かる．それに対し，lockset に関するハンドラも有効にすると，2 スレッドの監視でも 268.6%から 1452.6%（平均 780.0%），32 スレッドすべてを監視した場合には，1692.9%から 15461.1%（平均 4553.1%）ものオーバヘッドが生じることが分かる．

このように，director の監視内容によっては，実行時に非常に大きなオーバヘッドが生じる．そのような場合には，調査対象プログラムの実行全体を監視することは，あまり現実的ではない．表 6 に示されているように，監視する部分を絞り込むことにより，オーバヘッドを軽減することが可能である．

5.2 節で述べたように，本研究で構築したデバッガでは，利用者が調査対象プログラムの実行を制御しながら，特にエラーがあると思われるスレッドの，特定の実行区間のみに対し，data race の検出を行うことが可能である．このような機能は，director 利用者が，director のオーバヘッドをコントロールするために，非常に重要であると考えられる．

7. 関 連 研 究

本章では，提案環境と同様に director の開発に利用できる，既存の instrumentation ツールについて紹介する．またマルチスレッド以外の並行プログラミングの方式についても，簡単に紹介する．

7.1 Instrumentation ツール

提案環境以外にも，director の開発に利用できるフレームワークは多数提案されている．ここでは，提案環境と同様に，プログラムの instrumentation を行うものを紹介する[10),11),14),16),21)]．

まず instrumentation を対象プログラムの実行前（静的）に行うフレームワークとして，ATOM[21)]，

☆ 実行時間の比率の幾何平均より求めた．以下も同様である．

EEL[11]，Alamo[10] などがある．ATOM と EEL は，機械語コードの instrumentation を行うフレームワークである．これに対し，Alamo では C 言語のソースコードに対して instrumentation を行う．ATOM と EEL では，instrumentation を行うコード（たとえばコールバック関数を挿入するコード）を，ツール開発者がじかに記述する必要がある．これに対して Alamo では，event-driven のアーキテクチャを採用している．そのためツール開発者が監視したいイベントを指定すると，それに基づいて自動的に必要な instrumentation が行われる．

また instrumentation を対象プログラムの実行時（動的）に行うフレームワークとして，Valgrind[14] や DIOTA[16] などがある．これらのフレームワークでは，提案環境と同様に，仮想マシン上で機械語コードを実行しながら，必要な instrumentation を行っていく．またこれらのフレームワークは，マルチスレッド（Pthreads）にも対応している．特に Valgrind は，x86，AMD64，PPC32，PPC64 など複数のアーキテクチャ上で動作可能であり，比較的広く利用されているフレームワークである．Valgrind では，機械語コードを一度 UCode と呼ばれる中間言語に変換する．そして UCode に対して instrumentation を行ってから，再度機械語コードを出力する．ツール開発者は，UCode に対しじかに instrumentation を行うことや，監視したいイベントを指定してハンドラを登録することが可能である．

上述のフレームワークに共通している特徴として，主にプログラムの実行の監視の機能（instrumentation 部分）のみに特化していることがあげられる．そのためプログラムの実行の制御に関する機能は，非常に限定的である．また提案環境のように，実行の制御を行いながら，監視内容（instrumentation）を柔軟に変更していくこともできない．そのため，これらのフレームワークは，対象プログラムの実行を最初から最後まで通して監視し，最後にその監視結果を出力する形式の director の開発に向いていると考えられる．

しかし 6 章でも述べたように，プログラムの実行の制御と監視の機能を柔軟に組み合わせて利用することは，director のオーバヘッドをコントロールするために非常に重要である．そこで本研究では，そのような柔軟性を持ったフレームワークを特にプログラム実行制御・監視環境と呼び，上述のような監視機能のみに特化した instrumentation ツールと区別を行っている．

7.2 分散メモリ型並行プログラミング

本研究で特に着目したマルチスレッドは，並行プログラムの様々な実現方式の 1 つである．マルチスレッドと同様に広く利用されている方式として，メッセージパッシングに基づいた分散メモリ型の方式[13),22)] がある．

分散メモリ型の方式では，独立したアドレス空間を持つ複数のプロセスが，最小限のデータをメッセージとして送受信しながら，並行して処理を行う．マルチスレッドと異なり，各プロセスのアドレス空間が独立しているため，各プロセスを異なるコンピュータ上に分散して配置することが比較的容易である．そのため本方式は，多数のコンピュータを結合し非常に高い処理速度を得る，クラスタのような用途に向いていると考えられる．

マルチスレッドと同様に，分散メモリ型の方式においても，各プロセス間の相互作用による複雑性の増大が非常に大きな問題となる．そのため様々なツールやその構築方法に関する研究がなされているが，特に本研究に関連が深いと思われるものとして，OMIS（On-line Monitoring Interface Specification）[12] があげられる．OMIS 自体は，並行プログラムの監視を行うシステムと，それを利用するツール間のインタフェースを定めた仕様である．そして OMIS に準拠している監視システムは，OCM（OMIS Compliant Monitoring system）[24] と呼ばれる．OMIS のように，監視システムとツール間のインタフェースを標準化することには，2 つの利点がある．まずツール開発者は，OMIS によってしっかりと標準化されたインタフェースに沿って，開発を進めることができる．また OMIS に準拠したツールと監視システムは，非常に高い相互運用性を持つことができる．

8. ま と め

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に比べて非常に複雑である．そのような複雑な振舞いの調査には，director が非常に有用である．そこで本論文では，マルチスレッドプログラムに対応した director の開発のための，プログラム実行制御・監視環境を提案した．提案環境は，筆者らが以前の研究で提案した環境[26] を基盤として，以下の方針で設計を行った：(1) Pthreads API とできるだけ互換性を持たせる，(2) ユーザ空間スレッド機構を利用する，(3) スレッドごとに監視内容を細かく設定できるようにする．また本論文では，実際に提案環境を利用して構築した director の例として，data race 検出機能を持つデバッガを紹介し，その評価結果についても述べた．

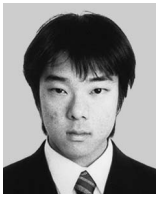
今後の課題としては，4.3.1 項で述べたプログラムの逆実行の機能[4] を，マルチスレッドプログラムに対しても利用できるようにすることなどがあげられる．

参 考 文 献

1) Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V.M. and Stichnoth, J.M.: Fast, effective code generation in a just-in-time Java compiler, *Proc. ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation*, pp.280–290 (1998).
2) 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).
3) Ball, T. and Larus, J.R.: Optimally profiling and tracing programs, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1319–1360 (1994).
4) 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).
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) 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).
7) Engelschall, R.S.: GNU Pth — The GNU Portable Threads. http://www.gnu.org/soft-ware/pth/
8) Harrow, J.J.: Runtime checking of multithreaded applications with Visual Threads, *Proc. 7th Int'l SPIN Workshop on SPIN Model Checking and Software Verification*, pp.331–342 (2000).
9) Hastings, R. and Joyce, B.: Purify: Fast detection of memory leaks and access errors, *Proc. Winter USENIX Conf.*, pp.125–136 (1992).
10) 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).
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) Ludwig, T., Wismuller, R., Sunderam, V. and Bode, A.: OMIS — On-line Monitoring Interface Specification (Version 2.0), Technical Report TUM-I9733, SFB-Bericht Nr.342/22/97A, Technical University Munich (1997).
13) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Technical Report UT-CS-94-230 (1994).
14) Nethercote, N. and Seward, J.: Valgrind: A program supervision framework, *Electronic Notes in Theoretical Computer Science*, Vol.89, No.2 (2003).
15) Nicbols, B., Buttlar, D., Farrell, J.P.（著），榊　正憲（訳）：Pthreads プログラミング，オライリー・ジャパン (1998).
16) 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).
17) Rosenberg, J.B.（著），吉川邦夫（訳）：デバッガの理論と実装，アスキー出版局 (1998).
18) Saito, Y.: Jockey: A user-space library for record-replay debugging, *Proc. 6th Int'l Symp. on Automated Analysis-driven Debugging*, pp.69–75 (2005).
19) Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs, *ACM Trans. Computer Systems*, Vol.15, No.4, pp.391–411 (1997).
20) Sosic, R.: Design and implementation of Dynascope, a directing platform for compiled programs, *Computing Systems*, Vol.8, No.2, pp.107–134 (1995).
21) 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).
22) Sunderam, V.S.: PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience*, Vol.2, No.4, pp.315–339 (1990).
23) The Open Group: The Single UNIX Specification, Version 2. http://www.opengroup.org/onlinepubs/007908799/index.html.
24) Wismuller, R., Trinitis, J. and Ludwig, T.: OCM — A monitoring system for interoperable tools, *Proc. SIGMETRICS Symp. on Parallel and distributed tools*, pp.1–9 (1998).
25) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 programs: characterization and methodological consider-

ations, *Proc. Int'l Symp. on Computer Architecture*, pp.24–36 (1995).
26) 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).
27) 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).

(平成 18 年 7 月 4 日受付)
(平成 18 年 12 月 6 日採録)



孝壽　俊彦（学生会員）

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学生会員．日本学術振興会特別研究員．

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 年慶應義塾大学理工学部情報工学科専任講師．2006 年より同大学助教授．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 各会会員．

土居　範久（名誉会員）

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，日本学術会議副会長．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」領域総括，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．