細粒度スレッド対応デバッガのポータブルな実装方式

大西勇次[†] 鎌田十三郎[††] 瀧和男[††]

細粒度スレッド機構は，高コストなプロセス生成や切替えを極力抑えるもので，多くの並列言語に実装されている．また，これらの並列言語の多くは高い移植性を獲得するため，既存の言語を介した2段組のコンパイル構成を採用している．本研究の目的は，細粒度スレッド機構を備えた並列言語に移植性の高いデバッガを提供することである．

このようなデバッガを実現するには，各処理系の細粒度スレッド機構に対応したスレッド管理部をポータブルに実装することが重要である．これらのスレッド機構の多くは，スタックアーキテクチャ上での効率的実行を目的として，遅延タスク生成などの技術を用いている．このため，既存デバッガによって得られる中間言語上のスレッド情報は，言語上のスレッド情報と対応していない．

本研究は，既存デバッガの改変を必要としない，細粒度スレッド対応デバッガのポータブルな実装法を提案する．本方式におけるスレッド管理は，言語上のスレッドの実行状況を把握するための少数のランタイムフックと，既存デバッガの実行を制御しスレッドのトレースを実現するインタフェース部から実現される．本方式の有効性は，並列言語 Cilk を対象とした実装例を通して確認した．

Portable Implementation of Debuggers for Fine-grain Multithreading Systems

Yuji Onishi,[†] Tomio Kamada[††] and Kazuo Taki[††]

Many concurrent language systems adopt their own fine-grain multithreading system to allow efficient execution of thread creation and context switch. Many of these language systems also adopt a compiler implementation technique to achieve portability that is two-step compilation framework. In these systems, a source code is usually transformed into C code, and then object code is produced by using an ordinary C compiler. Our goal is to provide a portable implementation of a symbolic debugger for these concurrent languages.

To provide such a debugger, it is important to portably implement the debugger component that manages their thread mechanism for the specific multithreading systems. These language systems usually adopt fine-grain multithreading systems that work efficiently on stack architecture using techniques such as lazy task creation. This implies that a thread in the underlying C language does not naively correspond to that of the target language.

This paper presents a portable implementation scheme of a symbolic debugger for fine-grain multithreading systems. This scheme does not require customization of the existing C debugger, instead supposes a few runtime hooks on the thread scheduler and an interface unit between the C debugger and the programmer. The runtime hooks are used to recognize thread status of the target language, and the interface unit controls the underlying C debugger to manage the program execution during tracing a thread. We show the effectiveness of our implementation scheme by applying it to a parallel language Cilk.

1. はじめに

並列プログラミングを行う際にスレッドを多数実行しても実行性能を損なわないように，スレッド生成・管理のオーバヘッドを抑制した細粒度マルチスレッド機構が多数提案されている．多くの並列言語が，その処理系に独自の細粒度スレッド機構を実装しており，プログラマはスレッド生成時のコストをあまり気にすることなく，必要に応じてスレッド生成を気軽に行うことができる．

現在，並列言語の中には単純なシンボリックデバッガさえ備えておらず最低限のデバッグ環境が整っていないものが多い．この原因は，言語処理系が実験段階にあるものが多いためで，また実装コストの高さの問

† 神戸大学大学院自然科学研究科
The Graduate School of Science and Technology, Kobe University
†† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University

題でもある．これは，デバッガを実装するにあたり，(a) 逐次言語への実装と同じくシンボル情報の対応をとる必要があるだけではなく，(b) 独自の細粒度マルチスレッド機構が行う複雑なスレッド管理に対応しなければならないからである．

本研究では，細粒度スレッド機構を備えた並列言語に対して，二段組コンパイル方式の特徴を利用したポータブルなシンボリックデバッガの実装法を提案する．二段組コンパイル方式とは，実行コードを生成する際に，プログラムのソースコードから機械語に直接コンパイルするのではなく，C 言語などの既存の言語（以下，単に C とする）に一旦コンパイルし，そこで生成されたコードを C コンパイラによって機械語にコンパイルするというもので，並列言語の実装コストを削減するために採用しているものが多い．この方式を利用することで，機種間で異なる関数呼び出しの慣例などを C のコンパイラで吸収することができ，また生成されたコードを C デバッガを用いてデバッグできる．我々が提案する実装法は，中間言語である C の既存のデバッガを利用するが，改造はしない．(a) C と 並列言語間のシンボル情報の差異は，C デバッガとユーザの間にインタフェースを設けて吸収する．加えて，インタフェースにおいて (b) 独自のスレッド管理機構が持つスレッド情報を理解し，対象とするスレッドの実行を管理する．我々は，スレッドの実行状態が変化する数箇所に限定してランタイムフックを挿入することで対象スレッドの管理を可能にした．このフックにより対象スレッドの状態をインタフェース側で認識することが可能で，これに応じてインタフェース側から既存デバッガを通してスレッド実行を制御することも出来る．これにより，細粒度スレッド機構が例えば Lazy Task Creation[5] のように，単一スタック上で複数のスレッド実行を管理している状況においても，対象スレッドの実行状況の把握ならびに実行管理を行うことが可能となる．本方式は，フックに要するコード量も少なく，インタフェースも単純に実装できるので，実装コストが低く，移植性も高い．

我々は Charles E. Leiserson らによって 開発された並列言語 Cilk[1),9)] に対して実装を行い，本方式の有効性を確認する．Cilk の処理系は変換後の C の上で適切な行番号情報などを保持しており，既に gdb などの既存のデバッガで正しいシンボル情報の対応をとることが可能である．我々は Cilk runtime にフックを埋め込み，加えて gdb 上のインタフェースを Emacs 上に構築することで，適切なスレッド管理を行うことのできるシンボリックデバッガの実装を行った．

2. 概　　観

2.1 提供するデバッガの機能

本節では，我々のデバッガの機能について述べる．

我々は「スレッドをトレースする」という言葉を使う．単に「スレッド」と呼ぶ場合，並列言語が定める論理的なスレッド（**細粒度スレッド**）を指す．プログラムの実行中，ユーザが特定のスレッドに着目してその挙動を追跡することを「スレッドをトレースする」と呼ぶ．これに対し，設定したブレイクポイントまでプログラムの全スレッドを自由に進行させることを継続実行と呼ぶ．我々のデバッガは (a) **継続実行**の機能と，指定スレッドのソース上での一行単位でのトレースである (b) **一行実行トレース**機能を提供する．(b) については，一行実行トレースが可能なスレッドは一時に一本と限定する．加えて，当該スレッドを停止または再開すると，別の実行可能スレッドもそれと同時に停止または再開する．つまり，スレッドのトレースは可能であるが，スレッドスケジューリングを制御する目的でトレース機能を使うことは出来ない．ユーザは，トレース中スレッド生成・終了命令においては，(c) **制御の流れに沿ったスレッド間にまたがるトレース**を行うこともできる．つまり，スレッド生成命令においてユーザが一行実行トレースを行う場合，ユーザは現在のスレッドを継続してトレースするか，新たに生成されたスレッド（子スレッド）をトレース対象にするか選択することができる．また，トレース対象のスレッドが終了する際は，そのスレッドの終了を待っているスレッドに移り，終了同期命令からトレースを再開することができる．

次に，スレッドの識別に関する対応について述べる．我々のデバッガは，プログラム実行に対して一意に定まる識別値（ID）を各スレッドに対して付加するのではなく，(d) **ユーザが実行時に指定する複数のスレッドに対してのみ ID を発行**する方法をとった．理由の 1 つは，このようなスレッド ID を提供している細粒度スレッド機構は少なく，また，デバッガ側で対応する場合でも処理系内部に少なからず手をいれる必要があり，ポータビリティを損なうからである．加えて，我々は全スレッドに対して ID をつける必要は少ないと考えている．数多くのスレッドがある中でユーザが特定のスレッドに着目するのは，呼出し関係などから容易に定まるスレッドを調べたいか，ブレイクポイントなどで捉えられたスレッドであるか，当該スレッドへの参照を入手した場合である．つまり，ユーザが途中で着目したスレッドに対して，必要に応じて印をつ

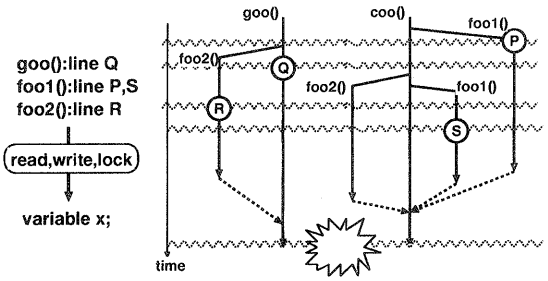


図 1 複数スレッドにまたがるデバッグ
Fig. 1 Debugging with multiple threads.

けることができ，スレッドのその後の挙動を必要に応じて確認（機能 c，f を使用）できれば，十分デバッグは出来ると考える．スレッド識別に関する議論を尽くすためには，プログラム再実行に関する議論も必要であるが，これは 5 章で行う．以下，ID の付加されたスレッドを「識別対象スレッド」，その中でも一行実行トレースの対象となっているスレッドを「トレース対象スレッド」と呼ぶ．

その他の機能としては，「info threads」とコマンド入力することで，(e) **存在する全スレッドの状態を表示する**ことができる．そして，この情報を使用して (f) **トレースするスレッドを明示的に別のスレッドに切り替える**こともできる．

このデバッガを用いて以下のような手順でデバッグを進めることが出来る．一般に，エラー原因の箇所を発見するには，逐次の時のデバッグと同様の操作を行う．即ち，エラーに関係すると思われる変数や構造体などにアクセスしているコードにブレイクポイントを設定したり，制御の流れに沿ってスレッドの挙動を順に追って行き意図しない挙動をしている箇所が無いか確認する．ブレイクポイントで捉えるべき状態を厳密に指定できない場合は，緩い条件でブレイクポイントを設定，停止状態に応じてユーザが取捨選択する．

具体例を通して説明をする．プログラムの再実行を繰り返しながらエラー原因をさかのぼって行くと，誤った値になっている変数 x があり，この変数 x に関係する部分が複数のスレッドに広がっていたとする（**図 1**）．変数 x に関係する行 (P, Q, R, S) を調べるためにブレイクポイントを設定し，プログラムを走らせたところ，図 1 の P で停止したとする．この時 P 上の各値はもちろん，「info threads」命令で他のスレッド情報を得ることもできる．また，ブレイクポイントで捉えたスレッドを一行ずつトレースしていくことも可能であり，goo もしくは coo を実行中のスレッドに対象スレッドの切り替え，一行実行トレースを行うこともできる．エラー原因を発見できなければ，継続実行を行い，次のブレイクポイント付近を調べることになる．図 1 の次は Q で停止したとする．この状態で「info threads」命令を行うと，4 つのスレッドが存在することが分かる．このように，ブレイクポイントやスレッドトレースの機能を用いて関係するスレッドの指定コード付近を順次調べて行くことができる．

このように，我々のデバッガは限定された機能だけを提供しているが，上で示したようなデバッグ手続きをとることで並列プログラムのデバッグを行うことができると考える．我々がこのようなデバッガの機能の限定を行ったのは，1 つにはデバッガ実装の際にポータビリティを損なわないようにしたいためであり，加えて，シンボリックデバッガの最小限の機能として十分であると考えたからである．これに関しては，プログラムの再演実行の話を含めて 5 章においてさらなる議論を行う．

2.2 デバッガを実現するための基本機能

2.1 節で述べたデバッガシステムを提供するには大きく分けて以下の 2 つ機能が必要である．

(1) 対象言語上での行番号，変数名，データ構造の情報の取得
(2) 指定スレッドの識別ならびにトレース制御

(1) の機能を実現するためには，C デバッガが提供する C コードのシンボル情報と並列言語のシンボル情報を対応させるだけで良い．この機能は細粒度スレッド機構に限定された話ではなく，すでに様々な技術が提案されている．一方，(2) の機能については，細粒度スレッド機構に特有の問題が存在する．本研究で対象とするのは (2) の機能の実現である．(2) の機能を細粒度スレッド機構に対して実現しようとする際に問題となる点を 2.3 節で明らかにし，それに対する解決策として我々のスレッド管理方法を 3 章で述べる．

2.3 細粒度スレッド機構とデバッガ実現上の問題

まず，細粒度スレッド機構についてその概要を述べる．細粒度スレッド機構には様々な種類があるが [10]，代表的なモデルである LTC (Lazy Task Creation) [5] を対象に議論を進める．

スレッドを実行するために必要な値や，実行すべきコードの番地などを保持するものを**フレーム**☆と呼ぶ．子スレッド生成時には新たなフレームが生成され，通常の関数呼出しと同じように当該プロセッサの現在のスタックの上に積まれることになる（**図 2**）．スレッ

☆ 通常の関数単位のものではなく，スレッド単位でフレームと呼んでいることに注意．

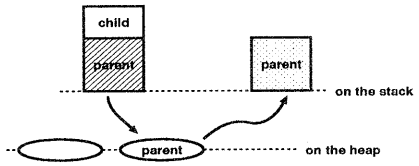


図 2 スタック上のフレームとヒープ上のスレッド
Fig. 2 Thread frames on a stack and on a heap.

ド生成を行ったプロセッサは，スタックトップにある子スレッドの実行に移り，元の位置にスタックトップが戻った時点で親スレッドが再実行される．

一方，フレームをヒープ上に生成する場合も存在する．例えば，別のプロセッサ上にスレッドを生成する場合，ヒープ上にフレームを確保して子スレッドのコンテクストを格納し，相手プロセッサのスケジューリングキューに挿入する．また，スタックトップ上で実行されていたスレッドがブロックされ，他のスレッドにスケジュールを移行したい場合，現在のスレッドのコンテクストをヒープ上のフレームに退避することになる．一度ヒープ上のフレームに退避された後，再び実行可能となった場合，フレームの内容に基づいて実行が再開されることになる．この時，元のプロセッサ上で実行が再開される必要は必ずしも無い．

ここで注目すべきは，このようなスレッド機構を C を用いて実装した場合，対象言語上のスレッドはあくまで論理的な意味においてスレッドとして扱われているのであり，実装上は OS の提供するスレッドやプロセスと 1 対 1 対応していないということである．このため，スレッドのトレースを行うためには，あるプロセッサの挙動をトレースするだけでは十分ではない．具体例を挙げて状況の説明を行う．

C に変換された並列言語のプログラムを C デバッガを用いてトレースしているとしよう．ユーザはあるスレッドのトレースを行っており，次に実行すべき命令が join（同期）であった．この時，まだ終了していない子スレッドが存在する場合，スケジューラは現在のスレッドの内容をヒープ上のフレームに退避し，新たなスレッドをスケジュールする．つまり，C デバッガを用いてプロセッサの実行をトレースしている場合，それまでトレースしていたスレッドは中断し，他のスレッドの実行状況が次に現れることになる．一方，ヒープ上に退避されていた親スレッドがその後実行可能となった場合，そのスレッドは別のプロセッサで再開される場合も考えられる．この場合，この親のスレッドの実行はあるプロセッサ上で実行されていたものが，一度中断され別のプロセッサ上で再開されたことになる．つまり，このスレッドの実行をトレースするにはこれら一連の状況を捉え，C デバッガを適切に制御して複数のプロセッサ間にまたがるトレースを行わなければならないことになる．

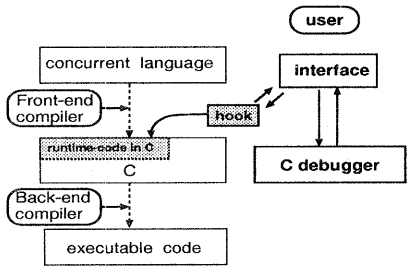


図 3 提案するポータブルなデバッガシステム
Fig. 3 The Overview of our debugging system.

3. スレッド管理

3.1 方　　針

2.3 節での問題を解決するためには，

- **対象スレッドの識別**
- **対象スレッドの中断・再開を捉えたトレース制御**

が重要である．本章では，本研究で行った解決方法を以下に述べる．**図 3** は，我々のデバッガシステムの概観である．

第一の要素であるスレッドの識別について，細粒度スレッド機構自身が不変的な ID を提供していることは少ない．また，スレッドの位置情報もスレッドの実行状態に応じて様々に変化する．このため，我々は識別対象スレッドに対して，常に最新の位置情報を保持し続けることでスレッドの識別を実現することにした（3.2 節）．対象スレッドの実行状態変化を捉えるために，同期操作などの実行スレッドが切り替わる可能性のある部分にランタイムフックを設ける．このフックでは，このランタイムを実行したスレッドが識別対象スレッドか判定し，必要に応じて位置情報の更新を行う．

一方，スレッドの実行状態の変化に応じて C デバッガを操作し，対象スレッドのトレースを管理するのは，図中のインタフェース部である．3.3 節において，インタフェース部がスレッドの実行状況に応じてどのように C デバッガを操作し，ユーザに対象スレッドのトレースを提供するのかを説明する．

3.4 節においては，これらの機構を実現する際に必要となる実装技術について議論する．ポイントの 1 つは，言語処理系にあまり手を加えずにランタイムフックを実現することであり，もう 1 つは，インタフェース部とランタイムフック部をどのように協調動作させ

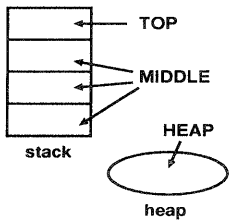


図 4 対象スレッドの 3 つの状態
Fig. 4 Thread Execution Status (TOP/MIDDLE/HEAP).

るかである．スレッドの実行状態が基本的にフック部のみが知り得るものである一方，デバッガの制御はインタフェース部のみが出来る操作であるため，両者の協調動作が必要となる．この協調動作を C デバッガを改造することなく容易に実現させる方式が望まれる．

3.2 スレッド識別機構

2.3 節でも述べたように，細粒度スレッド機構においてスレッドは以下の 2 通りの内部表現を用いて扱われる．即ち，通常の関数呼び出しと同じようにスタック上に実装されているフレームと，他のプロセッサ上で実行できるようにヒープ上に実装されているフレームである．我々は，状態変化しつつ実行されるスレッドをトレースするために，フレームの位置（location）情報を持つスレッド識別表を作成し，これを常に最新の状態にすべく管理する．位置情報は，スレッド状態とそれに応じた位置情報の組で表現される．また，スレッドがスタック上のフレームとして扱われている状況をさらに 2 つに分類し，対象スレッドの状態を以下のように 3 つに分ける（図 4）．

TOP 対象スレッドがスタックトップにおいて現在実行中であることを示す．状態とプロセッサ番号のみでスレッドを特定することが可能．この状態と次の MIDDLE の違いは，現在スケジュール中であるか否かの違いである．この 2 つを統合することも可能であるが，実装上の理由により区別する（後述）．

MIDDLE 対象スレッドがスタック上で待機中であることを示す．この場合，プロセッサ番号の他にスタック上でのフレームの深さ情報が必要になる．

HEAP 対象スレッドがヒープ上にフレームを保持していることを示す．この場合，ヒープフレームのアドレス（もしくは，それに準じるもの）を用いて識別することが可能である．

我々のデバッガは，現在トレース中のスレッドの状態を上のいずれかの形で保持している．我々のデバッガは複数の指定スレッドに対し識別機能を提供する（2.1 節の機能 d）ため，**表 1** のようなスレッド識別表を使用する．この表から識別対象スレッド 1 は，現在スタックトップにあり，プロセッサ番号は 0 番であることが分かる．スレッドの位置情報の更新は，言語処理系の中のスレッドスケジューリングに関係する部分にフックを設置して実現する（3.4 節）．

3.3 スレッドトレース管理

この節では，スレッドの実行をトレースする際に，対象スレッドの状態変化に対応してインタフェース部が行うべき C デバッガ操作について述べる．実装法に関する議論，つまりランタイムフックの挿入，及びフックとインタフェースとの協調動作の詳細については 3.4 節で議論する．

利用する C デバッガには以下の機能が備わっているものとする．これらの機能は，マルチスレッド対応の GNU debugger[8] や dbx といったデバッガによって提供されている．

- 特定プロセッサ上の一行実行トレース（関数の中に入るものを step 実行，そうでないものを next 実行と呼ぶ）
- 全プロセッサに対するブレイクポイントの設定
- 明示的なブレイクポイントのみ設定した状況でのプログラム実行（cont 実行と呼ぶ）
- デバッガ上からの値の読み書き，もしくは関数呼出し機能

インタフェース部は，ユーザからのコマンドやスレッドの状態変化情報を受けて，トレースの実現に必要な命令を C デバッガに対して発行する．

ユーザが指示するプログラム実行は，大きく 2 種類である．すなわち，継続実行と一行実行トレースである．継続実行の実現は容易である．インタフェースは，指示されたブレイクポイントを C デバッガを通して設定した後，C デバッガに対して cont 実行を指示するだけで良い．なんらかのブレイクポイントに衝突するまで，全スレッドが自由に実行される．

一方，一行実行トレースにおいては，トレース対象スレッドの識別（3.2 節）に加えて，スレッド実行状態の変化に即した C デバッガの操作が必要となる．以下で示すようにトレース対象スレッドの状態変化に応

表 1 スレッド識別表（識別対象スレッド 4 本の場合）
Table 1 Our thread identification table.

	ID	location		
		status	PE	depth or address
thread-1	1	TOP	0	—
thread-2	2	TOP	2	—
thread-3	3	MIDDLE	3	2
thread-4	4	HEAP	—	0x00022aff

じて C デバッガを操作することで，一行実行トレースを可能にする．

- TOP 状態：
  対象スレッドが TOP 状態である間，即ち，現在あるプロセッサで実行中である間は，そのスレッドの一行単位の実行トレースは，当該プロセッサの一行単位での実行トレースと同じである．
  つまり，インタフェース部は行対応を取りながら step/next 実行命令を C デバッガに対して発行する．
- MIDDLE 状態：
  スレッドが MIDDLE 状態となるのは，別スレッドが同じプロセッサのスタック上部で実行され，一時中断する場合である（TOP → MIDDLE）．スタック上層での活動が終了し TOP 状態として再開する（MIDDLE → TOP）まで実行を進める必要がある．C 上ではスタック上層での活動は普通，関数呼出しとして実装されるため，インタフェース部は TOP → MIDDLE 変化を捉え，C デバッガに対して next 実行を指示することで，スタック上部での活動をスキップ，対象スレッドのトレースを継続できる．但し，3.2 節で述べたように MIDDLE → HEAP 変化をする場合があるので，後述の対応が必要である．
- HEAP 状態：
  実行中（TOP）のスレッドが中断，HEAP 状態となった場合，その後スレッドが再スケジュールされる（HEAP → TOP）まで実行を進める必要がある．インタフェース部は，TOP → HEAP 変化を捕まえ，現在実行中の step/next 実行を解除した上で，C デバッガに対して cont 実行を指示する．その後，HEAP → TOP 変化が起こった時もその変化を捉え，cont 実行を解除する必要がある．その後は，TOP 状態としてスレッドトレースを継続できる．
- MIDDLE → HEAP への対応
  3.2 節で述べたように，一部のスケジューラはスタック上で停止中の対象スレッドをヒープ上に移動する場合がある．この際，インタフェース部は，C デバッガが実行中の next 命令を解除，cont 実行する必要がある．その後の操作は HEAP 状態と同様．

以上のように C デバッガを操作することで，指定スレッド上の実行トレースができる．これに加え，我々のデバッガは fork の際には子スレッドのトレースに移ることが可能であり，また，子スレッドのトレース終了時は親スレッドの join 部にトレースを継続することを可能にしている（2.1 節の機能 c）．この機能は，以下のようにして実現される．

FORK 時の子スレッドトレース 対象スレッド上の fork 命令においてユーザが子スレッドのトレースを希望している場合，以下のような操作を行う．

- 当該プロセッサ上で即座に子スレッドが実行される場合は，子スレッドのトレースに移る．C 上では普通関数呼出しとして実装されるため，子スレッドのトレースは，C デバッガの step 実行を行うことで実現できる．スレッド位置情報の更新は不要．
- 他のプロセッサ上にて実行を行う場合，まずヒープ上のフレームとしてスレッドを生成し（TOP →HEAP），その後，他のプロセッサで実行したと考える．即ち，子スレッドが HEAP 状態のトレース対象スレッドとして登録され，C デバッガに対しては cont 実行を指示する．

同期をまたいだトレース継続 トレース対象スレッド X が別スレッド Y と同期する場合，以下のようにして同期成立時点からトレースを再開できるようにする（join では，X が子スレッド，Y が親スレッドにあたる）．

- 親スレッド Y のフレーム上で子スレッド X が実行され，X の正常終了をもって同期の確認とする場合，スレッド Y の同期命令が実行されるまでプログラム実行を進める必要がある．スレッド位置情報の更新はともに TOP であるため必要無い．
  この場合，まず新たなトレース対象スレッド Y が join 命令までスキップ中であることを記録し，C デバッガに対し cont 実行を指示する．この間 Y は識別対象でもある．一方，join 命令側には，トレース対象スレッドが join 命令までスキップ中である場合を捉え，cont 実行を中断する仕組みを設ける．
- ヒープ上のオブジェクトを介して同期の確認を行う場合は以下の 2 通りに分かれる．
  スレッド Y が先に待ち状態になっている場合は，X が同期操作を行う際 Y の識別子をトレース対象として登録し，cont 実行を行う．Y が再スケジュールされた時点で，自動的にトレースが再開される．一方，Y がまだ到着していない場合は同期オブジェクトを記録して，C デバッガに対して cont 実行を指示する．一方，join 命令側には，同期対象のオブジェクトが先程記録したものである場合，Y が join を実行した時点でトレース対象に設定される仕組みを設ける．当然 join が成立すれば TOP 状態となり，さもなくば HEAP 状態となる．

このような機構を設けることで，fork/join といった同期イベントをまたぐ形でのスレッドトレースも可能にしている．3.2 節において，TOP/MIDDLE を別

状態としたのは，親子スレッドのトレースを移動する際にスレッド識別表の更新が複雑になるのを避けるためである．

3.4 実装技術

この節では，3.2 節で述べたスレッド識別表の更新，3.3 節で述べたスレッドトレースの実行管理をどのように実装するかについて述べる．3.3 節で述べた本研究の目的はポータブルにシンボリックデバッガを作成することであり，この実装方式が既存の C デバッガの改変を必要とせず，加えて言語処理系に対する改変を最小限に抑えることが重要である．

システム内の動作は大きく分けて以下の通り．

(1) ユーザコマンドの認識
(2) デバッガシステム用変数領域
(3) 状態変化のハンドルならびにスレッド位置情報の更新
(4) 状態変化に応じたインタフェースの起動
(5) C デバッガに対する実行指令

これらの動作の中にはランタイムフックを用いて実現する部分とインタフェース部の動作として実現する部分があるため，デバッガの挙動はフックとインタフェースの協調動作として実現されなければならない．

(1) は，インタフェースでのみ実現可能な動作である．

(2) には，コマンド情報などフック・インタフェースの両方から利用されるものや，スレッド識別表などインタフェース側からのみ利用されるものがある．このためすべて C の変数として実現を行い，フック・インタフェース側双方にアクセスを許す．

(3) は，言語処理系のランタイム部と密接に関わる部分であり，このためランタイムフックを挿入する形で実現する．ランタイムフックは，スレッドの中断，実行，同期を行う箇所，ならびに自動負荷分散機構がスタック上で中断している仕事を奪う箇所に挿入される．フックでは以下の動作を実現する．

- 自己スレッドが識別対象スレッドであるか確認，
- そうならば，対応する状態変化関数を呼び，
- スレッド識別表の当該位置情報を更新し，
- 必要に応じて，(4) インタフェース部の起動

次に，(4) インタフェース部の起動について述べる．これは，インタフェース側でしか実現できない機能である (5) を実行するために必要である．以下のようにして容易に実現することができる．ランタイム側にインタフェース部へ復帰するためのダミー関数を設け，インタフェースへの復帰を行う際は，この関数の呼出しを行う．インタフェース部は，予め C デバッガを用いてこのダミー関数にブレイクポイントを設定しておく．こうすることで，フックがダミー関数を呼び出せば，C デバッガが停止し，インタフェースに制御が復帰することになる．同時に，実行中の step/next 命令の解除も自動的に行われる．インタフェースに制御が復帰した後は，状況に応じた (5) の動作の実現は容易である．当然，これらの動作はユーザの目に触れない形で行う．

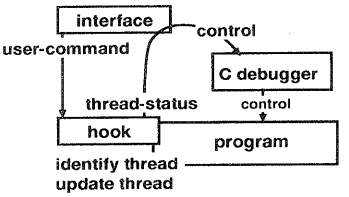


図 5 フックとインタフェースとの間の情報交換
Fig. 5 Information exchange between the runtime hooks and the interface unit.

このように，本方式ではデバッガの実現にあたって，ランタイム部分の一部改変（フック関数埋め込み）を行い，フックとインタフェースの協調動作により，既存の C デバッガに手をいれることなく，細粒度スレッドの実行トレースを可能としている．

4. Cilk への実装

本実装方式を用いて Charles E. Leiserson らによって 開発されている並列言語 Cilk[1),9)] に対して，シンボリックデバッガの実装を行った．Cilk の言語仕様は C の文法に並列処理用の命令を追加したものになっている．また，現在の実装では対象ハードウェアとして共有メモリマシンが想定されている．スレッド生成は，手続き呼び出しの前にキーワード spawn を付けることで指示される．spawn の意味は，親と子とが並列に実行を続けることを許すというものである．手続きは，sync 文が実行されるまで，子によって返された値を安全に利用することはできない．

我々が Cilk を対象言語として選択したのは，4.1 節で述べる先進的な実装上の特徴を備えているとともに，Cilk 処理系が既に Cilk と C のシンボル情報の対応関係を解決していたためである．つまり，本方式のスレッド管理機構を構築することで，容易にシンボリックデバッガの実現が出来る．Cilk 処理系で行われているシンボル情報の解決法は，以下の通りである．まず，Cilk と C はデータ構造や構文が共通なため，Cilk のソースの形を残したまま C のソースに変換される．残るは，行番号の対応であるが，これについては Cilk 処理系での行番号情報を，C preprocessor の機能を用いて C 上に残すことで実現されている．

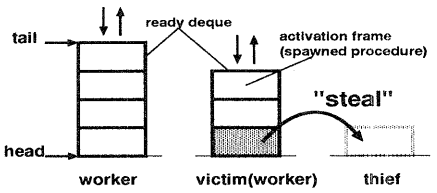


図 6 Cilk における work-steal 機構
Fig. 6 Work-stealing Mechanism of Cilk.

実装上の問題点の有無をランタイムフックの構築（4.2 節）と，インタフェース部の構築（4.3 節）の両面を通して議論する．また，ランタイムフックの構築に関しては，本方式を実現するに当たって言語処理系の実装者が支払うべき実装コストを検討する．

4.1 Cilk の細粒度スレッド機構

Cilk の細粒度スレッド機構は，基本的には 2.3 節で述べたスレッド管理方法を採用しており，加えて以下に挙げるような特有の先進的な機能を持っている．

two-clone 戦略 Cilk の Front-End コンパイラは，1 つの手続きに対して 2 つの clone（分身）を生成する．即ち，並列処理に対するサポートをほとんど行わずに高速動作する **fast clone** と並列処理に対する完全なサポートを行い，付随するオーバヘッドを被る **slow clone** である．

手続きが spawn されると，first clone が実行される．ただし，次に述べる work-stealing により手続きが盗まれた場合は，slow clone に変化して実行を継続する．プロセッサ間にまたがった同期は，slow clone によって実現される．このように同期のオーバヘッドを被る箇所を限定することで高速化を図ることが出来る．

work-stealing 負荷分散スケジューラの 1 つ．暇なプロセッサが，忙しいプロセッサからスレッドを「盗む」機構．**図 6** のように，spawn されたスレッドのフレームは，tail（＝スタックトップ）側で作業を行い，負荷分散を行う時は，head（＝スタックボトム）から仕事を盗む．他の負荷分散スケジューラとの大きな差は，Cilk が関数呼び出し的に実装されたスタック状のフレームから，仕事を奪うことができる点である．

4.2 フックの実装方式の検討

Cilk 上にフックの埋め込みを行うにあたりまず意識する必要があるのが，two-clone 戦略である．1 つの手続きに対して，2 つの関数が用意されるので，フックもこの 2 つの関数に対応したものを用意する必要がある．**図 7** に slow clone におけるランタイムフックの設置場所と疑似コードを示す．一方，fast clone では join でスレッドが切り替わることがないため，図中の join 周辺のフックは不要である．

スレッド識別表については以下のように実装した．Cilk version 5.2 の実装においては，フレーム固有のデータ構造として C の関数フレーム以外に別途，プロセッサごとに frame と呼ばれる構造体の配列が準備されている．このため，MIDDLE 状態のフレームの位置情報には各配列要素へのポインタを使用し，プロセッサ番号情報は不要であった．また，work-steal の際には，MIDDLE→HEAP に相当する識別表の更新が必要となる．一方，このような配列（frame）が準備されていない処理系での実装法を考えてみると，この場合は，スタックの深さを示すカウンタとプロセッサ番号を組として用いるか，スタックトップを示すポインタ値を使うことになると考えられる．

spawn と sync つまりスレッド同期の実現については，Cilk では fast clone を関数呼出しすることで spawn を実現している．sync の実現に関しては，fast clone と slow clone でその実現方式が異なっている．fast clone スレッドの終了は，対応する関数呼出しの終了をもって確認することが出来る．一方，slow clone に対するスレッド終了手続きは，最終的に slow clone に付随する closure 構造体（HEAP での識別単位）上の join counter を更新することによって実現される．これら spawn/sync の実装法に関しては，本研究で想定している細粒度スレッド実装方式の範疇に収まっており，我々のデバッガ実装方式をそのまま適用することが出来る．

次に，我々の実装方式のポータビリティについて議論する．**表 2** はランタイムフックの実現に当たって今回記述したコード量を示したものである．この実装では，識別対象スレッドであるかどうかの確認にハッシュ表を用いている．この表の中で，処理系実装者が新たに作った言語処理系に対して本デバッガを付加させようとした場合，必要となるのは，項目 c のマクロ呼出しを挿入する作業である．Cilk においては，高々 20 箇所 20 行程度であった．ここまでコード量が少ないのは，あくまで Cilk の実装において，スレッドの識別表の要素に既に実現されている変数値（frame）をそのまま使うことが出来たためである．つまり，識別表の適切な要素に出来合いの変数値を使うことが出来ない場合は，新たにスタックの深さ情報などを生成するコードが必要となり，変更箇所は当然増加することとなる．とはいえ，処理系に依らずほぼ共通に実装できるフック部と対応するインタフェース部をライブラリとして提供することができれば，言語開発者が容

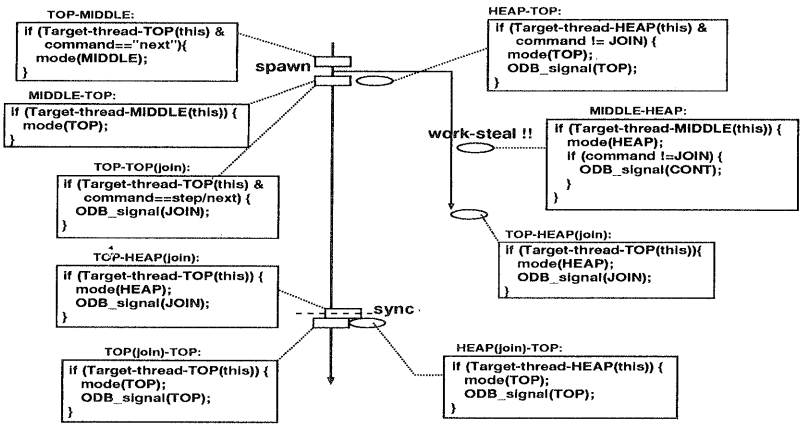


図 7 ランタイムフックの位置と疑似コード
Fig. 7 Runtime Hooks on the scheduler (pseudo code).

易にシンボリックデバッガ機能を付加することができるはずである.

最後に, 本方式の実行効率面での実用性について議論する. **表 3** は, これらフックを設置した場合に, プログラムの処理時間にどの程度影響するかについて, プログラム全体の実行時間を計測したものである. 使用したプログラムは, 処理系に添付されているフィボナッチ関数 (fib), N クイーン問題 (Q), ナップサック問題 (knap) である. hook の欄にフックつきのコードで実行した場合の時間を示し, normal の欄にフックをつけない通常のコードでの時間を示す. n は使用したプロセッサ数を示し, また, N はそれぞれの問題に使用した引数を示す. この表から分かることは, 我々のランタイムフックは fib の場合で高々5 % 程度, それ以外の問題では 2 % のオーバヘッドしか必要としてない. fib がプログラム実行中に占めるスレッド間同期の割合が非常に大きいことを考えると, この数字はシンボリックデバッガの実現にあたり十分許容出来る値である. つまり, 本方式は低い実装コストで十分な実行性能を得ることが出来たといえる.

4.3 インタフェースの実装方式の検討

インタフェース部は, 各種デバッガに対応した Emacs Lisp interface (gud.el) を改変する形で試験的実装を行っている. このプログラムは, Emacs 上で gdb などを起動した際, gdb と user の間にユーザインタフェースを設けるものである. ユーザからのコマンドを解釈し C デバッガ に送信する機能と, C デバッガ の返答を解釈し Emacs 上で表示する機能を備えている.

gdb を対象に以下のような実装を行った.

- ユーザコマンド解釈部の改造

「step/next/cont」の内容をランタイム側の変数にセットする. ランタイム・フックはこのユーザコマンド情報とスレッド識別の成否によって, 状態を変更するか, しないかを決定できる.

- インタフェースに制御を移すダミー関数の作成

ブレイク用のダミー関数を今回, `ODB_signal(int type)` として実装した. ランタイム側がインタフェースに発する情報の区別は type を用いて行っている. 現在, type は, 各フックの動作に応じて, 6 種類の

表 2 フックに要するコード量
Table 2 The number of the total code lines of our runtime hooks.

	種類 (言語処理系依存性)	行数
a	対象スレッド確認用マクロ 言語処理系によらず, ほぼ共通	80
b	変数宣言並びに状態更新処理 言語処理系によらず, ほぼ共通	50
c	処理系に埋め込まれた (a) の呼出 言語処理ごとに記述	20

表 3 フックを設置した場合としない場合での実行処理時間
Table 3 execution-time with hooks and without hooks.

func(N)	n	hook (s)	normal (s)	overhead
fib(30)	1	82.5	79.8	3.38%
fib(30)	2	41.4	39.6	4.54%
fib(30)	4	20.4	19.8	3.03%
fib(30)	8	10.6	10.4	1.92%
Q(18 × 18)	1	36.2	36.1	0.28%
Q(18 × 18)	2	36.3	35.6	1.97%
Q(18 × 18)	4	8.41	8.34	0.84%
Q(18 × 18)	8	8.52	8.45	0.83%
knap	1	181	180	0.55%
knap	2	97.9	97.8	1.02%
knap	4	46.7	46.2	1.08%
knap	8	22.8	22.4	0.44%

定数を割り当てている．インタフェースはこの定数をもとに，変化した対象スレッドの状態を知る．

- 通常ブレイクポイントとデバッガ制御用ブレイクポイントの判別
  上記ダミー関数と通常関数の区別を行い，通常関数の場合は，そのまま出力する．ダミー関数の場合は，定数 type により各種の処理を実行する．
- スレッド状態に応じたインタフェース部の挙動
  TOP になったという情報には，デバッガの制御をユーザに戻す．MIDDLE や HEAP になったという情報には，cont 実行をデバッガに指示し，次の状態変化を待つ．状態変化を待つ間は，Emacs 上の出力にフィルターを施し，ユーザに必要な情報のみを表示するようにする．

このようにインタフェースを実装することで，ユーザは通常の C デバッガを操作するのと同じ感覚で，細粒度スレッド機構を備えた並列言語のトレースを行うことができる．

5. 議　　論

並列プログラムのデバッガの研究としては，非決定性に関する話題，すなわちプログラムの再演実行[4]や競合検知[6]，加えてパフォーマンスチューニングツールなどに利用される実行可視化ツール[3]が挙げられる．本研究はこれらの研究とは異なり，複雑なスレッド管理を内部で行う細粒度スレッド機構に対し，単純なシンボリックデバッガをいかに手軽に実装するのかというものである．

関連研究：細粒度スレッド機構に対応したデバッガとしては，Multilisp[2]のデバッガである文献 3) がある．この中では，各スレッドに対して不変的な ID を付加している．これは，文献 3) が可視化を意図した研究であるため，当然の選択である．但し，実際にこのようなスレッド ID を付加するための拡張はスレッド生成・同期機構にそれなりの改変を必要とする．2.1 節で述べたように，我々は，ユーザが着目したスレッドに対してのみ ID を提供することで，より容易にシンボリックデバッガの提供を可能にしている．

また，再演デバッガとの関連については，我々のデバッガと再演機構の共存は可能である．再演デバッガは，モニタモードで記録された実行を再演モードで何度でも繰り返し実行を許すものである．モニタ実行時に実行順序に任意性のあるイベントについて順序関係のログをとり，再演実行時にそれを再現することで，同じ実行を繰り返す．基本的には再演実行時に変数アクセス命令やスレッド間同期命令に対し，実行順序を守るためのコードが付加された状態でプログラム実行が行われる[4]．但し，文献 7) の様に，実行環境の特性を利用し，スレッドスケジューリング順序を記録・再現することで実行順序の保証を行うものもある．いずれにせよ，再演実行用のランタイムに対して，今回用いたランタイムフックを挿入することで，再演実行プログラムに対して，我々のシンボリックデバッガ機能を付加することが出来る．

本デバッガの機能を制限した理由：本研究の目標は，最低限のシンボリックデバッガの機能を，ポータブルかつ許容範囲の実行効率で提供することである．

機能制限の 1 つとして，本デバッガは特定のスレッドの実行を停止したまま別のスレッドの実行をトレースすることはできない．この機能の実現にはスレッドスケジューリング機構に手をいれる必要がある．細粒度スレッド機構の中にはスレッドスイッチのタイミングを限定するなど独自の機構を備えたものが多く，実現が複雑になるため除外した．

次の機能制限は，全スレッドに対してスレッド ID を与えない点である．2.1 節の議論では，プログラムの再実行を含めた議論をしなかった．実は，バグ原因を求めて実行をさかのぼる際は，プログラムの再実行が行われる．この際，前回実行時に用いたスレッド ID を用いることが出来ればデバッグに有効である．しかし，注意したいのは，単にスレッド生成順に ID を付加しただけでは，スケジューリングに影響され，再実行時に有効でないという点である．実現にあたっては，スレッドの呼出し関係を保存するなどの対応が必要となる．

我々はスケジューリングや再実行対応のスレッド ID については，再演デバッガを併用することで解決すべきだと考える．加えて，再演機構は処理系・実行環境によって適したものが異なるため，分離して実現することが好ましいと考える．

6. ま と め

本研究では，細粒度マルチスレッド機構を備えた並列言語が 2 段組のコンパイラ構成を採用している場合に，シンボリックデバッガをポータブルに実装する方式を提案した．我々の方式は既存のデバッガを改造せずに利用し，必要最小限に挿入されるランタイムフックと上部インタフェースとの協調動作でデバッグシステムを実現する．加えて提案方式を実際に並列言語 Cilk に対して適用することで，実用上問題が無かったことを確認し，実装コストについて検討した．提案方式を用いてシンボリックデバッガを実装する際のコー

ド量は少なく，言語開発者にかかる負担は軽いと考える．加えて，言語独立に実装出来る部分をライブラリ化することで，その負担はさらに削減できると考える．これにより，提案方式の有効性が確認できた．

今後は細粒度スレッド機構を備えた他の並列言語への実装や細粒度スレッドライブラリに対して本方式が適用可能かどうか検討するとともに，本方式のライブラリの形での提供を目指してさらなる改良を行っていく予定である．

参考文献

1) Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. of PLDI*, pp. 212–223 (1998).
2) Halstead, Jr., R. H.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 501–538 (1985).
3) Halstead, Jr., R. H., Kranz, D. A. and Sobalvarro, P. G.: MULTVISION: A Tool for Visualizing Parallel Program Executions, *Proc. of Parallel Symbolic Computing*, LNCS, Vol. 748, pp. 183–204 (1992).
4) Leblanc, T. J. and Mellor-Crummey, J. M.: Debugging Parallel Programs with Instant Replay, *IEEE Transactions on Computers*, Vol. 36, No. 4, pp. 471–482 (1987).
5) Mohr, E., Kranz, D. A. and Halstead, Jr., R.H.: Lazy Task Creation : A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
6) Netzer, R. B. and Miller, B. P.: What are Race Conditions? Some Issues and Formalizations, *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, pp. 74–88 (1992).
7) Russinovich, M. and Cogswell, B.: Replay for concurrent non-deterministic shared-memory applications, *Proc. of PLDI*, pp. 258–266 (1996).
8) Stallman, R. M.: GDB マニュアル (1989).
9) Supercomputing Technologies Group: *Cilk-5.2(Beta1) Reference Manual*, MIT Lab. for Comp. Sci. (1998). `http://theory.lcs.mit.edu/~cilk`.
10) 田浦健次朗：細粒度マルチスレッディングのための言語処理系技術，コンピュータソフトウェア，Vol. 16, No. 2, pp. 1–19, No. 3, pp. 9–28 (1999).

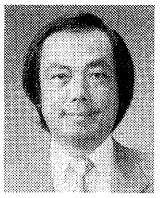
(平成 11 年 7 月 16 日受付)
(平成 11 年 12 月 29 日採録)

大西 勇次

1975 年生．1999 年神戸大学工学部情報知能工学科卒業．1999 年より同大学大学院自然科学研究科情報知能工学専攻博士前期課程に在籍．学士（工学）．並列・分散処理，言語処理系等に興味を持つ．

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996～1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．修士（理学）．並列・分散処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．



瀧 和男（正会員）

昭和 27 年生．昭和 51 年神戸大学工学部電子工学科卒業．昭和 54 年同大学院修士課程システム工学修了．工学博士．同年（株）日立製作所入社．昭和 57 年（財）新世代コンピュータ技術開発機構に出向．遂次型および並列型推論マシンと並列応用プログラムの研究開発に従事．平成 2 年同機構第 1 研究室室長．平成 4 年 9 月神戸大学工学部情報知能工学科助教授．平成 7 年 4 月同学科教授．LSI 設計技術と CAD，並列処理とマシンアーキテクチャ，脳型コンピュータ等に興味を持つ．電子情報通信学会，IEEE，ソフトウェア科学会，ACM，日本神経回路学会各会員．