2007-2-(6): 情報処理学会プログラミング研究会 発表資料 2007 年 8 月 1 日
(2007 年 12 月 16 日改訂)

L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構

八 杉 昌 宏[†1] 平 石 拓[†1,†2]
篠 原 丈 成[†1] 湯 淺 太 一[†1]

本研究では，実行中のプログラムがその実行スタックの内容を合法的に見たり変更したりするための “L-closure” という新しい言語機構を提案する．L-closure は入れ子関数定義を評価すると生成される軽量 lexical closure であり，lexical closure は生成時環境における lexical スコープの変数にアクセスできるため，その間接的な呼び出しにより合法的なスタックアクセスの手段を提供する．L-closure が追加された中間言語を高水準コンパイラで採用することにより，ごみ集め，チェックポインティング，マルチスレッディングや負荷分散といった高水準サービスがすっきりと効率よく実装可能となる．L-closure がアクセスする各変数では，共有のための場所のほかに，レジスタ割当て候補として私用の場所も用いる．共有の場所との値の一致をとるための処理や，L-closure の初期化処理は実際に L-closure が呼び出されるまで遅延される．多くの高水準サービスは L-closure を頻繁に生成するもののめったに（例えばごみ集めにおけるルートスキャンのために）呼び出さないので，総合的なオーバヘッドは大幅に削減可能となる．GNU C コンパイラ（GCC）は入れ子関数をもともとサポートしており，L-closure を実装するための GCC の拡張は比較的低コストで可能であった．本論文ではその実装の詳細を述べる．また，L-closure の低い生成・維持コストを示す評価結果について，変換ベース L-closure 実装についての追加評価結果とともに述べる．

L-Closures: A Language Mechanism for Implementing Efficient and Safe Programming Languages

Masahiro Yasugi,[†1] Tasuku Hiraishi,[†1,†2] Takenari Shinohara[†1]
and Taiichi Yuasa[†1]

We propose a new language mechanism called “L-closures” for a running program to legitimately inspect/modify the contents of its execution stack. L-closures are lightweight lexical closures created by evaluating nested function definitions. A lexical closure can access the lexically-scoped variables in the creation-time environment and indirect calls to it provide legitimate stack access. By using an intermediate language extended with L-closures in high-level compilers, we can implement high-level services such as garbage collection, checkpointing, multithreading and load balancing elegantly and efficiently. Each variable accessed by an L-closure uses a private location as a register allocation candidate other than a shared location. Operations to keep coherency with shared locations as well as operations to initialize L-closures are delayed until an L-closure is actually invoked. Because most high-level services create L-closures very frequently but call them infrequently (e.g., to scan roots in garbage collection), the total overhead can be reduced significantly. Since the GNU C compiler provides nested functions, we enhanced GCC at relatively low implementation costs. We present the implementation details. We also present the results of performance measurements which exhibit quite low costs of creating and maintaining L-closures, with additional experimental results on our transformation-based implementation of L-closures.

1. はじめに

様々な種類の計算機（マシン）について優れたコード生成器を実装するのは手間がかかる作業である．このため，高水準言語向けコンパイラの作成においては，ある程度，移植性が良くマシン独立な中間言語として C 言語を使うことも多い．つまり，高水準言語から C 言語への翻訳系（トランスレータ）のみを開発するのである．

C プログラムをコンパイルして得られる機械語プログラムの多くは，実行効率を考慮し，実行スタック

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
†2 日本学術振興会特別研究員
Research Fellow of the Japan Society for the Promotion of Science

を使う．関数呼び出しの際にはスタックフレームが割り当てられ，関数のパラメータや局所変数の他，戻り番地，1 つ前のフレームポインタ，callee セーブレジスタや `alloca` されたスペースのために使われる．高性能・高信頼プログラミング言語に備わってる高水準実行時サービスの中には，ごみ集め，自己デバッグ，スタックトレース，チェックポインティング，マイグレーション，継続，マルチスレッド，負荷分散などのように，その効率よいサポートのためには実行スタックの内容を見たり変更したりする必要があるものも多い．しかし，C 言語では，関数呼び出し中に呼び出されたほうの関数は，呼び出したほうの関数の局所変数に効率良くアクセスすることはできない．そのような局所変数のいくつかは callee セーブレジスタに値をとるようにできるかもしれないのだが，**ポインタ**に基づくアクセスはそのようなコンパイラの最適化技法の邪魔となる．さらに，スタックフレームのレイアウトはマシン依存であり，実行中のプログラム自身による偽造ポインタによる直接的スタック操作は本質的には不正である．実際，スタックフレームのデータはアプリケーションレベルのデータではなく実行のためのメタデータであるし，そのような不正なアクセスはセキュリティの問題もつながる．

例えばごみ集め（GC）を実装するためには，コレクタはすべての**ルート**を見つけられなくてはならない．各ルートは，ごみ集めされるヒープ中のオブジェクトへの参照を保持する．C では，呼び出し元におけるポインタ変数がオブジェクトへの参照を保持しているかもしれないが，それは，実行スタック中に眠ってしまっているかもしれない．たとえ直接的スタック操作を許すとしても，コレクタがスタック中のルート（参照）を他の要素から区別するのは難しい．**スタック上のマップ**というものが使えるかもしれないが，それは本来の C のデータでもないし，その準備にはコンパイラによる特別なサポートを必要とする．よって，通常，保守的コレクタ[1] がいくつかの制約下において使われることになる．一方，コピー GC を正しく実装するためには，コレクタは正確にすべてのルートをスキャンする必要がある．オブジェクトは空間の間を移動させられるため，すべてのルートポインタはオブジェクトの新しい場所を参照すべきだからである．正確なコピー GC は,「構造体とポインタ」に基づく翻訳手法[6),7)] によっても実現できるが，局所変数を構造体のフィールドへと翻訳すると，多くのコンパイラ最適化技法は使えなくなる．

このような問題を解決するために，強力で移植性の高い新しい中間言語が研究されている．例えば，C--[14),16)] は,（C より低水準に当たる）移植性の高いアセンブリ言語であり,「スタック歩き」するための実行時システムを提供することで，実行スタック中に眠る変数へのアクセスを可能としている．これにより，C--は，ごみ集めを含む高水準実行時サービスのいくつかを実現するための中間言語として適している．

本論文ではこのような中間言語として “L-closure” を伴う拡張 C 言語 XC-cube を新たに提案する．これにより，実行中のプログラムがその実行スタックの内容を合法的に見たり変更したりできる．具体的には，データ構造や変数の値としてアクセスすることを許す．L-closure は入れ子関数定義を評価すると生成される軽量 lexical closure であり，lexical closure は生成時環境における lexical スコープの変数にアクセスできるため，その間接的な呼び出しにより合法的なスタックアクセスの手段を提供する．C--と比較し，我々のアプローチでは，よりすっきりと高水準サービスがサポートされるだけでなく，C コンパイラ拡張で実装する場合，既存のコンパイラの大部分のモジュールやリンカなどの関係ツールを再利用することで，低コストでの処理系実装が可能となる．

本研究は，L-closure 機構の設計・提案を中心として，その (A) 実装研究，(B) 応用研究から成る．前者は，(A-1) C コンパイラ拡張方式の研究，(A-2) トランスレータ方式の研究から成る．(A-1) には L-closure のための具体構文の研究や GCC 拡張方式の研究が含まれる．(A-2) としては標準的な C 言語への手の込んだ翻訳方式を開発した．(B) は L-closure（または入れ子関数）を活用する応用技術（高性能・高信頼プログラミング言語に備える高水準実行時サービスの実装技術）を扱う．

これらの研究のうち，(B) の応用研究については，本論文では，基本的に関連研究[8),11),13),19),23),26)] として扱う．(A-2) は，(A-1) より後から始めた研究であるが，すでに発表[9),10)] に至っているため，本論文の主要な貢献とはいえない．ただし，本論文は本研究を総括し性能評価では (A-2) も対象とする．

本論文の主要な貢献となる (A-1) については，文献 25) において，C コンパイラ拡張方式の実装モデルの議論を行っているが，これに加えて本論文では GNU C コンパイラ（今回用いたのは GCC-3.2 ではなく GCC-3.4.6）に基づくコンパイラとしての実装の詳細も報告する．また，具体構文上の問題を解決し，(A-2) も対象としてより詳細な性能評価を行った．また，本論文では L-closure をより普遍的な概念（上位概念）

と明確に位置づけ，実際の用語の使用においては，文脈に応じて，適宜，特定の実装方式による下位概念，あるいは，具体的な低レベルのデータ構造を意味することとした．

以下，2 章では，本論文において続く章でも用いるサンプルプログラムを示す．3 章では，提案する言語機構である “closure” と “L-closure” の設計について示す．ここで，通常のトップレベルの関数との意味的互換性がない形で入れ子関数を用いることを提案する．4 章では，L-closure を C コンパイラ拡張で実装する場合の実装モデルを提案する．5 章では GNU C コンパイラに基づくコンパイラとしての実装の詳細を述べる．6 章では，別論文[9),10)] として発表済の，標準的な C 言語へと翻訳する変換ベースの L-closure 実装について紹介する．7 章では，L-closure の複数の実装（GCC 拡張実装と変換ベース実装）について性能評価を行い，L-closure の低い生成・維持コストを示す評価結果について議論する．8 章では，L-closure を持つ拡張 C 言語 XC-cube （または LW-SC 言語[9),10)]）への翻訳により種々の高水準サービスが実現できる点を含め，関連研究とともにその応用について議論する．9 章では L-closure のコストについて議論する．

2. サンプルプログラム

ある高水準言語のプログラムにおいて，再帰的に 2 分探索木のノードをトラバースし，対応する探索データを持つ連想リストを作成するものとしよう．そのような高水準言語プログラムは，**図 1** に示すような C プログラムへと翻訳されるとする．ここで，`getmem` は新しいオブジェクトをヒープ中に割り当てるものとし，コピー GC のコレクタが，`x`, `rest`, `a` や `kv` といったルートとなる変数をすべてスキャンできなくてはならないとする．もちろん，`bin2list` が再帰的に呼び出されているような状況も考える．

提案する中間言語 XC-cube では，コピー GC を伴うプログラムを**図 2** のように簡潔にすっきりとした表現で書くことができる．メモリアロケータ `getmem` は，入れ子関数定義を評価して生成される L-closure `scan1` を引数にとり，これを使うコピー型コレクタを起動するかもしれない．つまり，コピー型コレクタは `scan1` を間接呼び出しすることでルート（`x`，`rest`，`a`，`kv`）をスキャンしてオブジェクトの移動を行うとともに，さらに，入れ子状に L-closure `scan0` の間接呼び出しを行う[⋆1]．`scan0` の実体は呼び出し元における `scan1` の別のインスタンスかもしれない．スタックの底に達するまで L-closure の呼び出しを繰り返すことで実行スタック全体についてすべてのルートがスキャンできる．

図 2 において，`bin2list` の変数（`x`，`rest`，`a`，`kv`）は（callee セーブ）レジスタが割り当てられる可能性を持ってほしいが，よくある Pascal スタイルの実装を L-closure の実装として用いると `bin2list` におけるこれらの変数へのアクセスにレジスタ操作よりも遅いメモリ操作が必要になってしまう．というのは，`scan1` もまた，通常，静的リンクを通してスタックメモリ中のこれらの変数の値へとアクセスするためである．先に述べた「構造体とポインタ」に基づく翻訳手法[6),7)] でのスタック歩きでも同じ問題が起こる．

そこで，L-closure の新しい実装方針として，このような L-closure の**維持**コストを削減すること，すなわち，これらの変数へのレジスタ割当てを可能とすることを我々の目標とする．同時に，実装方針として，L-closure の**生成**コストも削減する．一方で，L-closure の**呼出し**コストは高くなっても構わないものとする．ごみ集めにおけるルートスキャンのためなど多くの高水準サービスにおいて，L-closure は頻繁に生成されつつ，たまにしか呼び出されないため，総合的なオーバヘッドをかなり削減することができる．

3. 言 語 設 計

Pascal や，多くの近代的なプログラミング言語（Lisp，Smalltalk，ML など）では，C とは異なり，関数中で定義される関数，つまり入れ子関数が許されている．本拡張 C 言語には，Pascal 型の入れ子関数を採用する．局所変数定義に実行点が至ると（論理的には）その変数の場所が作られるのと同様に，入れ子関数定義に実行点が至ると，入れ子関数本体とその環境のペアである lexical closure が（論理的に）生成される．入れ子関数は生成時の lexical スコープの変数にアクセスでき，lexical closure へのポインタは，lexical closure を間接呼び出しするために利用できる．lexical closure は変数の場所と同様にスタック上に作られるため，ごみ集めのある言語と異なり，lexical closure へのポインタは入れ子関数定義のあるブロックの実行完了後は使うことができない．

我々は，入れ子関数を通常のトップレベルの関数とは意味上別個に扱うことを提案する．これにより，入れ子関数に関しては異なる呼び出し列を用いて性能

⋆1 あるいは，`scan1` が `scan0` をリターンすることで，末尾呼び出しを除去することも考えられる．

```
Alist *bin2list(Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  if(x->right) rest = bin2list(x->right, rest);
  kv = getmem(&KVpair_d);              /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(&Alist_d);                /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(x->left, rest);
  return rest;
}
```

図 1　サンプルプログラム: 木–リスト変換
Fig. 1　A motivating example: tree-to-list conversion.

```
typedef void *(*move_f)(void *);

                /* scan0 is an L-closure pointer. */
Alist *bin2list(void (*scan0) lightweight (move_f),
                Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  void scan1 lightweight (move_f mv){ /* create L-closure */
    x = mv(x); rest = mv(rest);       /* scan roots */
    a = mv(a); kv = mv(kv);           /* scan roots */
    scan0(mv);                        /* scan older roots */
  } /* pass pointer to L-closure "scan1" on the following calls. */
  if(x->right) rest = bin2list(scan1, x->right, rest);
  kv = getmem(scan1, &KVpair_d);    /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(scan1, &Alist_d);      /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(scan1, x->left, rest);
  return rest;
}
```

図 2　L-closure（入れ子関数）での GC ルートスキャン
Fig. 2　Scanning GC roots with L-closures (nested functions).

改善に結びつけることができる．このため，**closure** という概念を導入する．これは通常の関数とほぼ同じように使えるが，通常の関数とはみなさないようにする．構文上は，キーワード `closure` を図 2 における `lightweight` と同様に用いる．closure ポインタを通常の関数ポインタとして，あるはその逆に使った場合は型エラーにする．

さらに，通常の関数とも，closure とも異なるものとして，軽量の closure である，**L-closure** という概念を導入する．そのための構文はキーワード `lightweight` を用いて図 2 のようにする．結果的に，提案する拡張 C 言語 XC-cube は，次の 2 種類の lexical closure を扱えるものとし，それぞれに，次の目的と制限を課すことにする．

Closure　は静的リンクを普通に用いる Pascal 型の実装を用いる．通常のトップレベルの関数との相互運用はしない．closure を所有する関数は，closure を維持するにあたり，変数がメモリにとられてしまうなど維持コストが必要となる．また生成コスト，呼び出しコストともそこそこのコストとなる．コンパイラ実装の代わりに,「構造体とポインタ」に基づく翻訳手法を用いてもよい．

L-closure　は，積極的に，その生成コストや維持コストをできるだけ最小化するという方針を採用する．その際，呼び出しコストは犠牲にしてよい．通常のトップレベルの関数や closure との相互運用はしない．L-closure を呼び出せるのは L-closure を所有する関数かその呼び出し先からのみとする．例えば，別スレッドからは呼び出せない．

状況に応じて適切なほうを用いればよい．例えば，2 章や 8 章で議論するような高水準サービスの実装では，L-closure を使うべきである．これらの L-closure は稀にしか呼び出されないので，生成・維持コストを最小化することが望ましい．

ここで，構文について考察しておく．我々は当初，`lightweight` を `inline` と同様に記憶クラスの一種と

```
/* L-closure 定義
     (“..” = “lightweight”) */
void f..(int x) { ··· }
/* L-closure を返す
    L-closure 定義 (構文のみ OK) */
void f..(int x)..(short) { ··· }
/* L-closure ポインタを返す
    L-closure 定義 */
void (*f..(int x))..(short) { ··· }
/* L-closure ポインタを受け取る
    関数定義 */
void f (void (*g)..(short)){ ··· }
/* または */
void f (void g..(short)){ ··· }
/* L-closure ポインタを受け取る
    関数プロトタイプ宣言 */
auto void f (void..(short));
```

図 3 提案する構文による例
Fig. 3 Examples based on the proposed syntax.

して L-closure 定義:

`lightweight void f (int x) { ··· }`

L-closure ポインタ型の変数定義:

`lightweight void (*f) (int);`

という構文を用いていたが，例えば，L-closure ポインタ型のフィールド宣言には使えないという問題や，返値型を L-closure ポインタ型とする L-closure 定義が表現できないなどの問題があった．もちろん，`typedef` を用いて，逐一，型名を与えていくことでこの問題は回避できるが，以下では，構文としての問題解決を与える．問題解決には，L-closure ポインタ型の変数定義を例にとると以下の 3 案が考えられる．

(A) `void lightweight (*f) (int);`

(B) `void (lightweight *f) (int);`

(C) `void (*f) lightweight (int);`

このうち，(C) 案を採用した．また，実際には `closure` の代わりにドット 1 個，`lightweight` の代わりにドット 2 個を用いて

(C’) `void (*f)..(int);`

のような表記も許すことにした．この構文は**図 3** のように問題なく使える．(A) 案では，L-closure を返す L-closure 定義（最終的には意味的なエラーになるが構文としては認められる）を表現できないなどの問題が残り，また，(B) 案では，L-closure ポインタを受け取る関数プロトタイプ宣言において “(*)” の部分の省略ができないなどの問題が残る．

4. 実装モデル

本章では，XC-cube の closure，L-closure それぞれについて C コンパイラ拡張により実装する場合に推奨される実装モデルを提案する．

closure については，スタック割り当てのポインタペアとして実装できる．入れ子関数本体とその環境（静的リンク）のペアである．closure ポインタはこのペアを指すようにすればよい．closure ポインタを使って間接呼び出しをするときには，コンパイル時にすでにポインタの型により通常の関数ポインタとは区別されているため，コンパイルされた呼び出し列としては，静的リンク（ペアの後者）をセットしてから入れ子関数本体（ペアの前者）を呼び出すようにする．ここで，closure を 2 要素の構造体のように扱い，値としての一括コピーを許すといった設計も考えられるが，「関数名や配列名はポインタに成り下がる」という C 言語の仕様を closure にも採用している．これにより，関数の場合との構文の違いは，キーワードを付加の有無だけになる．なお，すべて直接呼出しだった場合はペアから取り出さなくても入れ子関数本体と静的リンクの値は分かるため，ペアのスタック割り当ては不要とできる．

L-closure ポインタも closure と同様のペアを指すとするが，L-closure の生成コストを最小化するため，L-closure の初期化（ペアの初期化）は実際に呼び出されるまで遅延させる．つまり，生成コストは事実上 0 とすることになる．これは，注意深く実装された例外ハンドラと同様である．

L-closure の維持コストを最小化するためには，もし，関数 f が L-closure 型の入れ子関数 g を所有していて g は f の局所変数（もしくはパラメータ）x にアクセスするのであれば，x は場所を 2 つ用いることにする．具体的には，共有のための場所のほかに，(callee セーブレジスタの）レジスタ割当て候補として私用の場所も用いる．この分離により，既存の最適化器やレジスタ割り当てへの干渉をできるだけ減らす．ここで，もし x のアドレスがとられている場合や，x にアクセスする入れ子関数が L-closure 型ではなかった場合， x は私用の場所を使わないことにする．また，入れ子関数 g がさらに入れ子関数 g_2 を持つときは，g_2 による f の変数へのアクセスは，g_2 の種別にかかわらず，g によるアクセスとみなす．

同種の手法は入れ子関数を持てると仮定した C 言語で，**図 4** のように表現できる．(後述の「遅延された」前処理や後処理を含まない不完全な形ではある.) ここは図 2 の関数 `bin2list` を手本としている．関数 `bin2list` は，入れ子関数 `scan1` を所有しているが，(`p_x` や `p_a` といった）私用の変数を導入し，関数呼び出し部分を除いて普段は私用の変数を用いる．関数呼び出しの際には，`bin2list` は私用の値を（`x` や `a`

```
Alist *bin2list(void (*scan0)(move_f), Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  void scan1(move_f mv){               /* nested function */
    x = mv(x); rest = mv(rest);        /* scan roots */
    a = mv(a); kv = mv(kv);            /* scan roots */
    scan0(mv);                         /* scan older roots */
  } /* pass pointer to "scan1" on the following calls. */
  /* private variables */
  Bintree *p_x = x, Alist *p_rest = rest, *p_a = a; KVpair *p_kv = kv;
  if(p_x->right){
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = bin2list(scan1, p_x->right, p_rest);
    p_x = x, p_rest = rest, p_a = a, p_kv = kv,  /* post-processing */
    p_rest = _r;
  }
  {
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    KVpair *_r = getmem(scan1, &KVpair_d);       /* allocation */
    p_x = x, p_rest = rest, p_a = a, p_kv = kv;  /* post-processing */
    p_kv = _r;
  }
  p_kv->key = p_x->key; p_kv->val = p_x->val;
  {
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = getmem(scan1, &Alist_d);         /* allocation */
    p_x = x, p_rest = rest, p_a = a, p_kv = kv;  /* post-processing */
    p_a = _r;
  }
  p_a->kv = p_kv; p_a->cdr = p_rest;
  p_rest = p_a;
  if(p_x->left){
    x = p_x, rest = p_rest, a = p_a, kv = p_kv;  /* pre-processing */
    Alist *_r = bin2list(scan1, p_x->left, p_rest);
    p_x = x, p_rest = rest, p_a = a, p_kv = kv,  /* post-processing */
    p_rest = _r;
  }
  return p_rest;
}
```

図 4 C レベルでの私用変数と前処理と後処理の追加．説明のためだけのプログラムであり実際のコードとは異なる

Fig. 4 Adding private variables and pre-processing and post-processing in C. This is not the real code but shown for explanation purpose only.

といった）共有変数に**前処理**として保存する．同様に，制御がリターンされるときには，**後処理**として共有変数から私用の値を読み込む．本手法の結果，私用変数はレジスタ割り当て候補となれる．ただし，`scan1` が実際には呼び出されることがないとしても前処理と後処理が省略できないという問題が残る．このときの前処理や後処理を含む制御の流れを gc 関数からの `scan1` 呼び出し時において図に示すと**図 5** のようになる．

この問題に対処するため，我々は，**図 6** に示すような**遅延された前処理**と**条件付後処理**を提案する．前処理（図では■）は実際の L-closure の呼び出しまで遅延され，後処理（図では◆）は前処理において動的に有効化される．前処理は，(1) すべての L-closure の初期化（入れ子関数ポインタと静的リンクのペアの初期化），(2) 私用の値の共有場所への保存，(3) 後処理の有効化（戻り番地の変更による）のステップからなる．前処理が 2 回以上実行されてしまうこと[*1]を防ぐには，条件付後処理がすでに有効になっているかを確認すればよい．後処理は，単純に共有場所から私用の値を読み込めばよい．

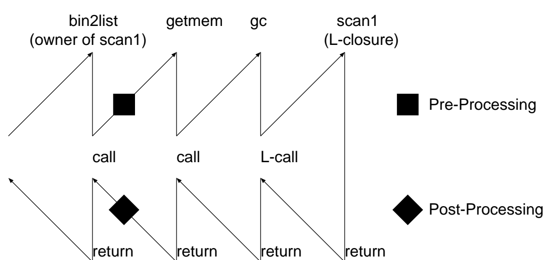

図 5 特に工夫しない前処理と後処理

Fig. 5 Usual pre-processing and post-processing.

⋆1 L-closure の再帰呼び出しなどで．

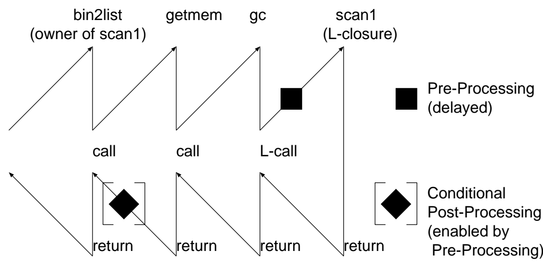

図 6 遅延された前処理と条件付後処理．L-closure が実際に呼び出されたときのみ

Fig. 6 Delayed pre-processing and conditional post-processing performed only if the L-closure is actually called.

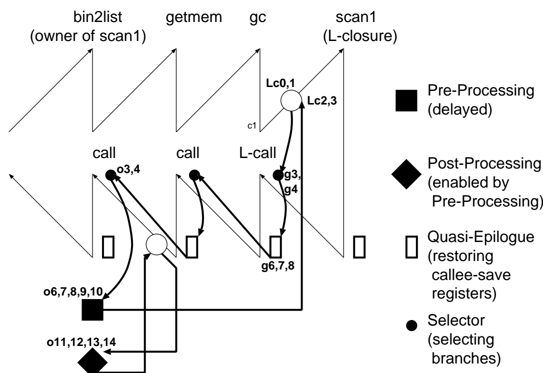

図 7 呼び出そうとしている正しい前処理のための，L-closure 所有関数への（非局所的）一時リターン．付加された数字は図 8 にある数字と対応

Fig. 7 (Non-local) temporary return to the owner of the L-closure to be called for correct pre-processing. Annotated numbers correspond to those in Fig. 8

実際に図 6 の■の時点まで前処理を遅延させるというのは，そう単純な話ではない．私用場所への callee セーブレジスタの割り当てが成功していた場合には，前処理 (2) が考える共有場所へ保存すべき私用の値というのはそのレジスタに保持されていてほしい．しかし，前処理をしようという図 6 の■の時点では，`bin2list` の呼び出し先（callee）である `getmem` や，その呼び出し先である `gc` などは，callee としてレジスタの値をセーブした上で，別の用途にそのレジスタを使ってよいとされているからである．正しい前処理のためには，先に callee セーブレジスタを元の状態に戻す必要がある．この際，スタックポインタも戻したのではリターンしたことと同じになってしまうので，フレームポインタのみを（他の callee セーブレジスタとともに）戻す．元の状態に戻すのは，**図 7** に示すように，所有する関数までの一時的な非局所リターンの間に特別な擬似エピローグを実行していくことで実現できる．

図 7 のための擬似コードを**図 8** に示す．これらの図

`bin2list`: // owner of `scan1`
`o0` : ...
`o1` : call `getmem` with selector `o3`.
`o2` : ...
`o3` : /* **selector** for `o1` */
`o4` : if (L-closure to be called is in this frame) jump to pre-processing `o6`.
`o5` : else jump to quasi-epilogue `o18`.
`o6` : /* **pre-processing** for `o1` */
`o7` : copy values from private locations to shared locations.
`o8` : initialize all L-closures (function-pointers and static chains).
`o9` : save and modify `o1`'s return address to enable post-processing `o11`.
`o10` : continue the L-call.
`o11` : /* **post-processing** for `o1` */
`o12` : save the return value.
`o13` : copy values from shared locations to private locations.
`o14` : continue the actual return with saved return address/value.
`o15` : ...
`o16` : /* **selector** for modified return addresses */
`o17` : continue the L-call.
`o18` : /* **quasi-epilogue** */
`o19` : restore callee-save registers.
`o20` : temp-return to the selector for the call in previous frame.

`gc`: // caller of `scan1` (= *scan*)
`g0` : ...
`g1` : L-call *scan* with selector `g3`.
`g2` : ...
`g3` : /* **selector** for `g1` */
`g4` : jump to quasi-epilogue `g6`.
`g5` : ...
`g6` : /* **quasi-epilogue** */
`g7` : restore callee-save registers.
`g8` : temp-return to the selector for the call in previous frame.

L-call f:
`Lc0` : save f and registers.
`Lc1` : temp-return to the selector for the call in previous frame.
`Lc2` : restore f and registers.
`Lc3` : setup f`->static_chain` and jump to f`->code`.

図 8 擬似コードと呼び出しステップ

Fig. 8 pseudo code and calling steps.

において，L-closure `scan1` の呼び出し（図 8 中 `g1` において `scan1` の **L-call** ）で，所有する関数への非局所的一時的リターンが開始される（`Lc0`, `Lc1`）．最初に直前のフレームのための**セレクタ**に一時的にリターンする．(例えば，`gc` のためには `g3`）．各セレクタ（例えば `o3`）はもしも，“現在の” フレームが呼び出そうとしている L-closure の所有者であった場合には前処理への分岐を選択する．(例えば `o4`→`o6`）それ以外の場合には擬似エピローグへの分岐（例えば `o5`→`o18`）を選択する．ここで，L-closure を全く所有しない関数の場合は，必ず擬似エピローグへの分岐が選ばれ（例

えば g3, g4, g6）, callee セーブレジスタ復元後に非局所的一時的リターンが続けられる（例えば g7, g8）.

前処理（o6,o7,o8,o9）は復元された callee セーブレジスタと "現在の" フレームを用いて実行できる．前処理後は，制御を実際の（"今" 初期化されたばかりの）L-closure へと渡すことになる.（o10, Lc2, Lc3）.

各一時的リターンにおいては，直前のフレームでの対応する呼び出しのためのセレクタを，戻り番地に基づき見つけられる必要がある．戻り番地を変更して後処理を有効にした後は，元のセレクタは見つけられない代わりに，前処理を重ねて実行することなく L-call を続けるためのセレクタが見つかるようにしている.（o16, o17）.

図 7 での scan1 の L-call のための実線矢印はくわしくは，図 8 の以下のステップに対応している: g1, L-call（Lc0, Lc1）, g1 用セレクタ（g3, g4）, 擬似エピローグ（g6, g7, g8）, . . ., o1 用セレクタ（o3, o4）, 前処理（o6, o7, o8, o9, o10）, L-call（Lc2, Lc3）, scan1.

図 7 は有効化された後処理が通常のリターンをインターセプトする点も表す. bin2list へのリターンのため実線矢印は，次のステップに対応している．getmem, 後処理（o11, o12, o13, o14）, o1, o2.

以上の L-closure に関する実装方針により，L-closure はそれを所有する関数から効果的に距離をとることができ，所有する関数による変数アクセスを高速化できる.

本章で述べた実装モデルは closure や L-closure を所有することにより，プログラムの実行時間がどういう影響を受けるかを見積る材料とできる.

5. GCC に基づく実装

本章では，GNU C コンパイラ[17] に基づく，コンパイラとしての実装の詳細を報告する．GCC-3.4.6 を強化することで，IA-32 と SPARC をターゲットマシンとして XC-cube の closure と L-closure を実装した.

GCC は生成しようとしているコードを表現するのに RTL（Register Transfer Language）という言語による中間表現を用いる．この表現形式は C よりはアセンブリ言語に近いものである．RTL 表現は抽象構文木から生成された後，様々なパス（データフロー解析，各種最適化，レジスタ割り当て）で変形された後，アセンブリコードへと変換される．同じプログラムからでもターゲットマシンが異なれば，生成される RTL 表現も異なるが，RTL の意味はほぼマシン独立となっている.

GCC では，C への拡張として独自の入れ子関数が使える．GCC の入れ子関数は通常のトップレベルの関数との相互運用性を確保するために,「トランポリン」という技法を用いている[2]．トランポリンとは静的リンクとして必要な環境をセットしてから入れ子関数本体のコードへジャンプする数命令の短いコードのことであり, スタック上に動的に生成される．トランポリンのコードアドレスを関数ポインタとすることによって通常のトップレベルの関数との相互運用性が確保される．ただし，(1) スタック上にコードを動的に生成することや，(2) アーキテクチャによってはプロセッサの持つ命令キャッシュを明示的にフラッシュする必要があること，(3) OS がスタックに実行可能なコードを置くことを制限している場合は，その制限の解除[*1]，という高い生成コストが発生する.

XC-cube の closure の実装は，すでに 4 章で述べた通りとした．ちょうど GCC が入れ子関数を実現するのに用いているトランポリンの代わりに，静的リンクとコードアドレスというポインタペアを用いることになる．GCC では入れ子関数の実装のためにすでに静的リンクなどにも対応しているので，例えば closure の呼び出し時には GCC 内部で static_chain_rtx の値である RTL の式（各マシンの特定のレジスタ等に対応）を静的リンク用追加引数と考えて呼び出し時に環境を設定すればよい．これらは,（コンパイラフロントエンドを含む）RTL 生成方式を拡張するだけで実装が可能であった.

一方, L-closure は, (1) RTL 生成方式の拡張, (2) アセンブリコード生成方式の拡張，(3) 短いアセンブリコード（実行時ライブラリ）の準備，によって実装した．実装指針として，実装の分かりやすさが犠牲になったとしても，できるだけ再利用性（移植性）や効率を高めるようにした．再利用性についていえば，既存の RTL を変更・拡張せずにそのまま用いることにした．これにより，ほどんどの既存の最適化器を変更・拡張せずに済ませられる．また，アセンブリコード生成方式の拡張は最小限として，できるだけ RTL 生成方式の拡張で対応するようにした.

以下，各節において，L-closure の GCC 実装の詳細について述べる前に，実装概略を述べておく．セレクタと擬似エピローグについては，単純にアセンブリコードを生成する．これは，既存のエピローグ生成部をお手本として実現できた．なお，今回，SPARC で

⋆1 制限が解除できない場合も考えられる.

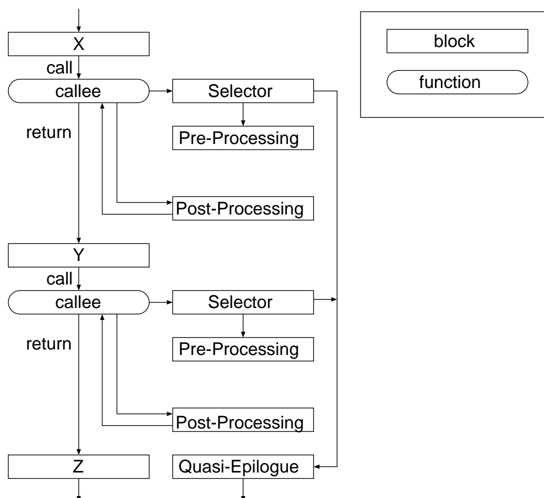

図 9 実装目標: 理想的な制御フローグラフ．L-closure の所有関数における 2 回の関数呼び出しに関して

Fig. 9 Implementation goal: ideal control flow graph for two function calls (for an owner function of L-closures).

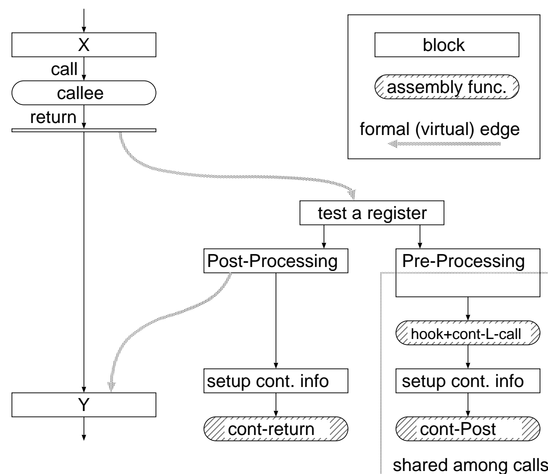

図 10 遅延された前処理と条件付後処理を含む関数呼び出しのための RTL 中間表現の制御フローグラフ

Fig. 10 Control flow graph of RTL representation for a function call with delayed pre-processing and conditional post-processing.

はレジスタウィンドウを使う実装とした．前処理・後処理のコードは RTL 表現として生成されるようにした．これは，通常の最適化やレジスタ割り当ての対象となる．セレクタから前処理コードへの制御の移動，後処理コードから乗っ取られた本来のリターン処理への制御の移動，といったアセンブリコードレベルでの制御の移動を反映した最適化を RTL レベルで正しく行うために，**仮想的制御フローエッジ**（仮想エッジと略す）を導入した．また，実際の実装では，関数単位で，そのすべてのセレクタを，1 個のマクロセレクタに結合している．マクロセレクタにより，結合される前のセレクタにより可能だった分岐すべてが可能になるようにしている．さらに，関数内部の異なる呼び出し点に関する複数の前処理や後処理のコードを部分的に共有するとともに，前処理や後処理や擬似エピローグに含まれる共通の処理を他の関数とともに共有するために短いアセンブリコードとしてライブラリ化することで，分かりにくくなるという代償は伴うもののコードサイズを抑制している．

5.1 RTL における仮想的制御フロー表現

図 7 や図 8 で示した擬似コードを，コントロールフローグラフの形で示すと，**図 9** のようになる．図 9 は L-closure を所有する関数について，その中で 2 つの関数呼び出しを含む形で，呼び出し毎の前処理，後処理，セレクタや，関数の擬似エピローグを示している．RTL レベルでの中間表現を含めて，図 9 の通りに実装できれば理想的で分かりやすいが，現実問題として次の問題があるため，実際の実装はもっと複雑にしている．

- 図 9 の `callee` は，通常の入口（呼び出し），出口（リターン）以外に，追加の 2 つの出口（セレクタへ，後処理へ），追加の 1 つの入り口（後処理から）がある．RTL には制御の移動先を表すラベルがあるが，ラベルが 3 つ追加された特殊な call 命令は標準の RTL では表現できない．また，そのような特殊な命令を追加した場合は既存の最適化器など（特に制御の流れを重視するデータフロー解析器）への修正も必要となる．
- RTL レベルで表現できない点もあるため，アセンブリレベルで実際の制御の移動を実装しているところもある．RTL の中間表現にはこれらの「後で生成またはリンクされるアセンブリコード」の内容を示すものは現れないが，それでも最適化などを正しく行うために，実際の制御移動を反映した仮想的な制御移動を RTL レベルで表現する必要がある．
- コード量を削減するため，共通の処理はコード共有するようにしたい．例えば，前処理において，L-closure を初期化する処理は共通である．

以上の課題を踏まえた検討の結果，**図 10** のコントロールフローグラフになるような（仮想的な）RTL レベルの中間表現を用いることにした．まず，実際には使われるが RTL レベルでは表現されないアセンブリコードは中間表現には現れていない（セレクタ，擬似エピローグなど）．アセンブリレベルでしか表現で

きない共通の処理のうち関数によらず共有するものは図 10 の「assembly func.」の呼び出しの形で表現する．ただし，`cont-return` と `cont-Post` はともにリターンしない関数として呼び出される．いわゆる**継続**[⋆1]が保存されているようにして，`cont-return` や `cont-Post` 内部ではそれらの継続を呼び出すようになっている．`hook+cont-L-call` についても基本的には L-call の続きをするため本来はリターンしないが，後処理のためにリターンをインターセプトする役割と戻り値を保存する役割[⋆2]を `hook+cont-L-call` の後半以降に持たせている（GCC 内部では untyped_call と呼ぶ）．なお，コード量を削減するため，図 10 で，前処理の後半以降の RTL コードは同じ関数内のすべての呼び出しで共有している．このため `cont-Post` により改めてそれぞれの呼び出し用の後処理に進む必要があるが，この「各呼出し毎に別の前処理または後処理に進む必要がある」という部分も，前処理と後処理に関して共有し，「test a register」を使って前処理と後処理に分岐するようにしている．

アセンブリレベルでの実際の制御の移動を反映した RTL レベルでの仮想的な制御の移動には，仮想的制御フローエッジ（図 10 中 formal (virtual) edge）の導入を提案する．この RTL 制御命令からは通常の RTL のようにはアセンブリ命令は生成されない．RTL 制御命令があった場所には，アセンブリレベルではコメントが入るように変換することにする．この仮想エッジは図 10 のように呼び出し命令の直後から前処理と後処理に先行する「test a register」に対してと，後処理から本来の呼び出し命令の直後（図 10 では Y ブロックの先頭）に対して用いている．これより，ある変数が前処理ブロックもしくは後処理ブロックの先頭で「生きている」なら，呼び出し命令の直後でも「生きている」し，ある変数が Y ブロックで使用されているなど Y ブロックの先頭で「生きている」なら，その先行節となる後処理ブロックの末尾でも「生きている」ことになる．この仮想エッジにより，最適化が意図したとおり正しく行われることになり，さらに言えば，前処理ブロックや後処理ブロックも最適化の対象になる．例えば，付録 A.1 の関数を IA-32 向けにコンパイルをした場合には，“`pc = 2`” が，前処理における pc の私用場所から共有場所へのコピーへと「定数伝播」され，値 2 を直接共有場所へコピーする命令へと置き換えられるとともに，“`pc = 2`” に関する処理は通常の実行パスからは削除された．

現在の実装では RTL の拡張を避けるために，仮想エッジを既存の RTL の枠組みで次のようにトリッキーに表現している[⋆3]：

```
(set (pc) (if_then_else
            (gt (pc) (const_int 0))
            (label_ref LABEL) (pc)))
```

ここでは潜在的に *LABEL* への分岐が存在しているように思わせることがポイントであり，実際にこの RTL 式の「RTL としての意味」[⋆4]を追求してはいけない．また，仮想エッジに相当する実際の制御移動が行われるのは L-closure を呼び出したときだけであるため，分岐確率（RTL では `BR_PROB`）を 0 に設定している．アセンブリコード生成では，

```
(set (pc)
     (if_then_else
       (gt (pc) (match_operand 1 "" ""))
       (label_ref (match_operand 0 "" ""))
       (pc)))
```

というパターンに対して補助関数を呼び出し，アセンブリ言語のコメント（または条件部の反転や then と else が交換される書き換えがなされたときは，無条件分岐命令）が生成されるように `define_insn`[17] で定義している．

実は，仮想エッジは，後のアセンブリコード生成において，マクロセレクタも利用する「各呼出し毎に別の前処理または後処理に進むコード」を生成するための情報を集めるのにも利用している．これは，図 10 では呼び出し命令の直後から「test a register」に対する仮想エッジであるが，他の仮想エッジと区別するために，`(const_int 0)` を `(const_int 1)` に変更したものを使っている．

5.2 実際の制御移動の実装

図 10 の RTL 中間表現は**図 11** に示すようなアセンブリコードへとコンパイルされる．図 11 は，L-closure を呼び出したときの実際の制御の流れも同時に実線太矢印にて示している．また，仮想エッジはアセンブリレベルでは削除される．

図 7 や図 8 で示した擬似コードや図 9 では，呼び出し（戻り番地）に対応するセレクタが 1 つという理想的なコードを示していたが，実際には，関数内のすべてのセレクタを結合したマクロセレクタを用いた．マクロセレクタは 図 11 中の Selector1 と Selector2

⋆1 一級ではない．
⋆2 実際には RTL で展開．
⋆3 実際には，4 つのバリエーションがある．
⋆4 プログラムカウンタ（pc）が 0 より大きければ *LABEL* への分岐

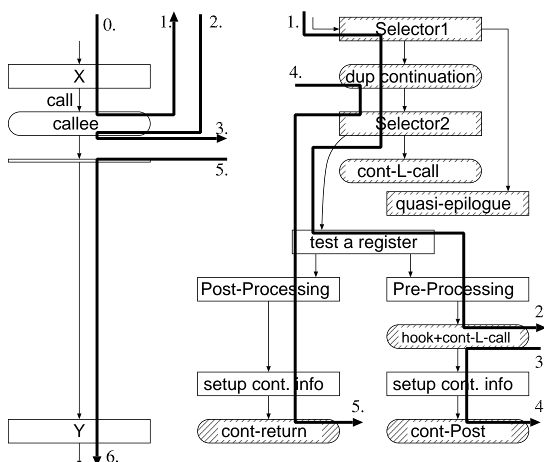

図 11 コンパイルされたコード上の実際の制御の移動
Fig. 11 Actual control-transfer over compiled code.

がその主要なパートである．Selector1 では擬似エピローグに進むかどうかを，呼び出そうとしている L-closure のアドレスが "現在の" スタックフレーム中なのかに基づき判定する．擬似エピローグに進まないときには Selector2 で戻り番地に基づき各呼び出し用の前処理に進むか，前処理済の L-closure 呼び出しを `cont-L-call` により続けるかを選択する．実際には Selector2 は,「各呼出し毎に別の前処理または後処理に進むコード」にもなっていて，現在の呼び出しに対応する後処理に進むためにも利用される．後処理へ進むのでも利用するために Selector2 の直前の「dup continuation」で（矢印 4 → 5 の 4 からの）継続を保存するとともに， Selector2 の後の呼出し毎の「test a regiter」で前処理と後処理に分岐するようにしている．この Selector2 の分岐先は，呼び出し命令の直後から「test a register」に対する，アセンブリレベルでは削除される仮想エッジの情報に基づいている．

以上で基本的なメカニズムを説明したので，実際に L-closure を呼び出すときの制御の流れを図 11 の実線太矢印に沿って確認しておく．関数フレームが今呼び出そうとしている L-closure を所有していなかった場合は擬似エピローグから，呼び出し元の Selector1（1 → 2 の開始地点）に制御が移される．さきほど説明したマクロセレクタの機能を使って前処理まで進み，前処理も終えると `hook+cont-L-call` にて呼び出そうとしている L-closure へと制御を移す.（1 → 2 の 2.）このとき `hook+cont-L-call` 内では自分の戻り番地（3 → 4 の 3）で，`callee` の戻り番地を（元の戻り番地を保存した後に）上書きしておく．その結果，`callee` からのリターンはインターセプトされ，矢印 3 → 4 の 3 へとトラップされることになる．この 3 では，まず `callee` 用の返値をスタックに確保した作業域に保存してから，現在の呼び出しに対応する後処理へと進むために `cont-Post` へと進む（3 → 4）. `cont-Post` 内部では,「dup continuation」で保存済の 4. からの継続を使い Selector2 の機能によって現在の呼び出しに対応する後処理へと進んだ後，本来のリターン処理のために `cont-return` へと進む（4 → 5）. 本来の戻り番地（5 → 6 の 5）に制御を移す前には保存しておいた `callee` 用の返値をレジスタ等にセットする．

後処理を有効化するのに使った，戻り番地の入れ替えは，例えば，戻り番地が制御できる Lazy Threads[5] や，ごみ集め処理を細分化するためのリターン・バリア[27] などにも見られる．本実装の面白い点としては，すでに戻り番地が入れ替えられているかどうかで後処理が有効になっているか判定できるため，他に空間的・時間的オーバヘッドを支払ってフラグを設けなくても，前処理を二重に行わないなどの判定が可能であり，しかもそれは Selector2 でマッチする戻り番地が見つからない場合の受け皿は `cont-L-call` であるというシンプルな形で実装できる．

5.3 L-closure の呼び出し

L-closure の呼び出しは，図 8 の `Lc0`–`Lc3` のような擬似コードの内容を持つアセンブリコードとして実装された `L-call` ルーチンの呼び出しとして実装されている．`L-call` ルーチンは，呼び出したい L-closure のポインタ f と実引数を受け取る．ここで，実引数は通常の呼び出し慣例により受け取る．また f については，`static_chain_rtx` で表される静的リンク用のレジスタ等で受け取るようにした．

5.4 セレクタの見つけ方

擬似エピローグは，呼び出し元の（マクロ）セレクタに制御を渡す．つまり，呼び出し元から呼び出し先へ，通常の戻り番地に加えてセレクタアドレスの情報を伝えたことになっていなくてはならない．単純にいえば，各関数呼び出しにおいて，複数の戻り番地が実質渡されていればよい．例えば，Lazy Threads[5] や C--[14),16] では特別な呼び出し慣例を使われており，追加の例外的な戻り番地は通常の戻り番地を基準とした単純な演算や埋め込んでおいた表により得られるようにしている．また，例外処理の典型的実装や StackThreads/MP[20] では，呼び出し慣例は標準的なものとしたまま，プログラム全体を通して通常の戻り番地から特殊な戻り番地への対応表を作成し，その中を探索する方式が使われている．

本処理系でも上で述べた方式を使うことはできる．おそらく呼び出し慣例を変えずに済むようにプログラム全体を通した対応表を作成することが望ましいと思われる．ただ現在は，プログラム全体を通した表を（リンカなどの助けを得て）作成せずに済み，なおかつ，呼び出し慣例も変えなくて済む方法を使っている．それは，マクロセレクタの先頭を特別な命令列で印付けしておき，通常の戻り番地から印が見つかるまでマシン命令をスキャンするという方法である．この方式はかなりトリッキーではあるが，本処理系はコンパイラとして実現しているため，asm 文や定数表など印と見間違う可能性があるものがコードに紛れ込む可能性があれば，それより前にセレクタへのジャンプを印付で挿入すればよいので，危険な方法ではない．性能面でも，プログラム全体を通した表を線形探索するのと比べて特別悪いとは考えにくい．

5.5　擬似エピローグ

擬似エピローグは，callee セーブレジスタやフレームポインタの値を呼び出し元が使えるように復元するが，スタックポインタの値はそのままにとどめるためのコードである．この手法は，マルチスレッドを実現するための StackThreads/MP[20] という先行研究で使われている．StackThreads/MP では，GCC が出力したアセンブリコードに対する後処理により擬似エピローグを得ている．一方，我々は，既存のコンパイラのエピローグ生成部を参考にしてコンパイラにより擬似エピローグのアセンブリコードを生成している．我々は，このほうが結果的に実装コストも低くできるものと考えている．

SPARC ではレジスタウィンドウも使われているため，L-closure の呼び出し時には，まずは図 8 の `Lc1` においてレジスタウィンドウのフラッシュ（スタックメモリへの書き出し）を行ってから，各擬似エピローグではレジスタウィンドウは回転させずにスタックメモリから callee セーブレジスタの値を復元する．

5.6　コンパイル例

L-closure を所有する関数 `cpfib` を付録 A.1 に示す．この関数 `cpfib` をアセンブリコードへと（`-O2 -mtune=ultrasparc3` にて）コンパイルした結果を付録 A.2 に示す（`scan1` は省略）．21 行目の call に対応して 29,67,68,70 行目のストア命令（`st`），13 行目の call に対応して 38,47,48,49 行目のストア命令で私用の値を共有場所に保存している（前処理の一部）．一方，21 行目の call に対応して 31 行目のロード命令（`ld`），13 行目の call に対応して 39,40,41 行目のロード命令で共有場所から私用の値を読み込んでいる

表 1　GCC へのパッチとしての実装コスト
Table 1　Implementation costs as patches to GCC

Closures	L-Closures + Closures		(lines)
RTL	RTL	IA-32(`i386`)	SPARC(`sparc`)
406 lines	1037	217 + 105asm.	181 + 132asm.

（後処理）．51–54,56 行目で，L-closure `save1` が初期化されている．16，42 行目のコメントや 24, 30 行目の無条件ジャンプ命令直後のコードへの流れる形はアセンブリレベルでは削除された仮想エッジの痕跡である．代わりに，82 行目や 84 行目からのジャンプ（前処理か後処理へ分岐するためのレジスタ`%o0` テスト命令へのジャンプ）や，後処理後の保存しておいた戻り番地への（`cont_return` 中での）ジャンプにより，実際の制御移動は実現される．

元の `fib` 関数が SPARC では 13 命令（`-march=pentium4` では可変長命令 72 バイト）であるのに対し，付録 A.1 の形での入れ子関数による高水準サービスの追加後の `cpfib` は，`scan1` のコードとあわせて，GCC では 82 命令（同 229B），closure では 63 命令（同 225B），L-closure では 97 命令（同 341B）となった．L-closure では命令数が増えるが，付録 A.2 に示すように，関数 `cpfib` の主要部分に対応する 5–25, 32–34 行目ではストア命令やロード命令を使わずにレジスタだけで高速に計算していることがわかる．メモリアクセス命令は，L-closure の初期化，前処理，後処理には必要だがこの稀にしか実行されない部分は主要部分の外に追い出す形になっている．また，前処理後半部以降のコード共有の効果は，call 数 n に対し，$10n - 17$ 命令（同，約 $41n - 59$B）の削減となっている．

5.7　実 装 結 果

GCC がすでに独自の C 拡張の入れ子関数を提供していることもあり，XC-cube の closure や L-closure の実装は比較的低コストで可能であった．

表 1 には GCC へのパッチとして XC-cube の closure のみ，または closure と L-closure の両方を実現したと仮定して実装コストを見積る場合の，パッチの行数を示す．例えば，IA-32 における L-closure の実装には，RTL コード生成について 1037 行のパッチ，アセンブリコード生成について 217 行のパッチを用いた．また，アセンブリ言語で直接作成したコード 105 行の追加が必要であった．このうち RTL コード生成部分については SPARC と共通である．

無変更の GNU デバッガを使用しての，L-closure を含む生成コードのデバッグには特に深刻な問題は発生していない．例えば，バックトレースはうまくいく．

ただし，いくつかの実行ステータスは正しく得られないことがある．

6. 変換に基づく L-closure の実装

前章までの成果は，文献 25) ならびに本論文の主要な貢献であるが，その研究発表の準備を進める過程で，標準的な C 言語へと翻訳する変換に基づいても L-closure の実装は可能であり，性能目標もある程度達成できるという着想を得た．当時，我々は高水準言語から中間言語 XC-cube へのトランスレータの作成を支援するために，S 式ベース C 言語（拡張/標準 SC 言語）を入出力とする SC 言語処理系[8] の開発を開始していた．この SC 言語処理系を L-closure を持つ拡張 SC 言語 LW-SC の変換ベースの実装のためにも利用することにした．(ただし，当時は LW-SC 言語が持つのは L-closure とは考えていなかった.）この研究成果は本論文に先立つ形で発表[9),10] に至っている．これにより普遍概念としての L-closure は複数の実装を持つこととなった．

SC 言語処理系については，その後も改良[11] を続けているが，この章では，LW-SC 言語の変換ベースの実装の概要を紹介しておく．以下で，明示的スタックとしているものは実際にメモリにとられるが，C のスタックに含まれるとしている局所変数などは実際にはレジスタ割り当ての対象となるように変換を行っており，コンパイラベース実装と同様に維持コスト削減を行っている．

まず，LW-SC の入れ子関数の定義は，C の親関数（入れ子関数定義を持っていた関数）の本体に残しておくことはできないので，すべてトップレベルに移動させる．ただし，単純にトップレベルに移動させただけでは，入れ子関数から親関数の局所変数にアクセスすることができない．そこで，入れ子関数の呼び出し時に親関数のフレームポインタを渡すようにする．このフレームポインタの値は，親関数の実行時に，通常の関数ポインタとの組で保存するようにしておく．

しかし，このフレームポインタを C の実行スタック上のフレームへのポインタとすることは（C のレベルでは）できない．そこで，メモリ上に C のスタックとは別の実行スタック（明示的スタック）を用意し，フレームポインタはこの明示的スタック上のフレームへのポインタとする．

プログラムの実行時，明示的スタックは C のスタックに含まれる局所変数と同じ内容を記憶するが，このスタックの値の更新・参照を頻繁に行うとメモリへのアクセスが増え，大幅な速度低下を招く．そこで，明示的スタックの値の更新・参照を行うのは入れ子関数の呼び出しのときのみとし，その他の局所変数の操作は基本的にすべて C のスタック上で行う．

入れ子関数の呼び出しが発生すると，実際にその関数に処理を移す前に，明示的スタック上の局所変数の値を C のスタック上の値で更新しておくようにする．これにより，親関数の局所変数には上記のフレームポインタを通してアクセスすることができるようになる．入れ子関数の実行終了時には，局所変数の値の変更を反映させるため，明示的スタック上の値を再び C のスタックに書き戻すようにしておく．

ただしこの明示的スタックの更新を行うためには，C のスタックの底に眠っている局所変数の値も参照しなければばならず，そのために C のスタックを一旦巻き戻す必要がある．そして入れ子関数の実行が終わったあと，C のスタックの状態やプログラムの実行位置などを呼び出し前の状態に戻す．

7. 性能測定

入れ子関数を使わない通常の C プログラムの実行速度は，拡張したコンパイラであっても変わらない．よって，lexical closure の生成・維持コストを測定するために，まずは，高水準サービスのための入れ子関数を伴うプログラムと，それを省いた標準の C プログラムとの比較を行った．また，L-closure の複数の実装について性能評価を行うため，6 章で述べた SC 処理系による L-closure の実装についても評価した．

また，生成・維持コスト削減のためには L-closure の呼出しコストは高くなっても構わないものとしたが，実際にどの程度の呼び出しコストなのかを人工的な例で測定した．

BinTree (copying GC) は，コピー GC されるヒープにおいて，200,000 ノードの 2 分探索木を生成する．

Bin2List (copying GC) は，コピー GC されるヒープにおいて，500,000 ノードの 2 分木を線形リストに変換する．(図 1 参照)

fib(37) (check-pointing) は，チェックポインティングのためのスタック状態キャプチャ機能つきで，37 番目の Fibonacci 数を再帰的に求める．(8.1 節参照)

nqueens(13) (load balancing) は遅延分割型負荷分散（8.3 節[13),23),26] 参照）のための機能つきで， N クイーン問題（N=13）のすべての解をバックトラック探索で求める． 部分解はその場更新し，バックトラック時には更新を一時的に取

表 2 性能測定 (生成・維持コスト) (1/2)
Table 2 Performance measurements (for creation and maintenance cost). (1/2).

S: UltraSPARCIII
P: Pentium4
X: Xeon
C: Core2Duo

Elapsed time in seconds
(relative time to plain C)

		no closures	trampolines	closures	L-closures(SC)	L-closures
BinTree copying GC	S(GCC)	0.191(1.00)	0.225(1.18)	0.215(1.13)	0.191(0.999)	0.183(0.961)
	P(GCC)	0.146(1.00)	0.156(1.07)	0.156(1.06)	0.152(1.04)	0.145(0.991)
	P(ICC)	0.132(0.904)	—	—	0.138(0.941)	—
	X(GCC)	0.154(1.00)	0.170(1.11)	0.166(1.08)	0.161(1.05)	0.158(1.03)
	X(ICC)	0.153(0.995)	—	—	0.154(1.00)	—
	C(GCC)	0.0814(1.00)	0.0894(1.10)	0.0905(1.11)	0.0842(1.03)	0.0833(1.02)
	C(ICC)	0.0788(0.969)	—	—	0.0803(0.987)	—
Bin2List copying GC	S(GCC)	0.295(1.00)	0.329(1.11)	0.294(0.997)	0.307(1.04)	0.293(0.993)
	P(GCC)	0.146(1.00)	0.149(1.02)	0.148(1.01)	0.153(1.05)	0.149(1.02)
	P(ICC)	0.100(0.685)	—	—	0.110(0.751)	—
	X(GCC)	0.160(1.00)	0.163(1.02)	0.163(1.02)	0.167(1.04)	0.163(1.02)
	X(ICC)	0.120(0.752)	—	—	0.130(0.810)	—
	C(GCC)	0.100(1.00)	0.104(1.04)	0.104(1.03)	0.108(1.08)	0.103(1.03)
	C(ICC)	0.0795(0.793)	—	—	0.0814(0.812)	—
fib(37) check-pointing	S(GCC)	0.888(1.00)	3.87(4.36)	1.39(1.56)	1.13(1.27)	0.990(1.12)
	P(GCC)	0.285(1.00)	0.552(1.94)	0.406(1.42)	0.549(1.93)	0.329(1.15)
	P(ICC)	0.291(1.02)	—	—	0.482(1.69)	—
	X(GCC)	0.335(1.00)	0.636(1.90)	0.468(1.40)	0.608(1.81)	0.394(1.17)
	X(ICC)	0.330(0.986)	—	—	0.577(1.72)	—
	C(GCC)	0.435(1.00)	0.570(1.31)	0.464(1.07)	0.712(1.64)	0.497(1.14)
	C(ICC)	0.343(0.789)	—	—	0.381(0.875)	—
nqueens(13) load balancing	S(GCC)	0.471(1.00)	0.935(1.98)	0.747(1.58)	0.589(1.25)	0.570(1.21)
	P(GCC)	0.318(1.00)	0.436(1.37)	0.443(1.39)	0.422(1.33)	0.452(1.42)
	P(ICC)	0.317(0.995)	—	—	0.482(1.51)	—
	X(GCC)	0.228(1.00)	0.339(1.49)	0.382(1.68)	0.350(1.53)	0.322(1.41)
	X(ICC)	0.266(1.17)	—	—	0.403(1.77)	—
	C(GCC)	0.257(1.00)	0.305(1.19)	0.357(1.39)	0.354(1.38)	0.315(1.23)
	C(ICC)	0.267(1.04)	—	—	0.375(1.46)	—

り消すがその記述にも入れ子関数を用いる．

図 1 や付録 A.1 に示したとおり，これらの高水準サービスのサポートのために，各関数は入れ子関数定義を所有する．つまり，呼出し毎に（すくなくとも論理的には）lexical closure が生成される．ここで，この測定では入れ子関数は呼び出されないようにした．つまり，ごみ集め，チェックポインティング，タスク生成は起動されない．

さらに，開発中の分散環境にも対応した遅延分割型負荷分散（load balancing）フレームワーク（8.3 節[11),13)] 参照）において，次のプログラムも用いた．Pentomino については，タスク分割の際のその場更新の部分解に対するバックトラック時の更新一時的取り消しにも，入れ子関数を用いる．

fib(37) 再帰による Fibonacci 数の計算であり，fib(3) すらタスク分割の候補とする細粒度計算を行う．

Comp(20000) サイズ N の配列の 2 要素間の比較を行う（N=20000）．データサイズ $O(N)$，計算量 $O(N^2)$ である．

Pentomino Pentomino パズルのすべての解をバックトラック探索で求める．

LU(1000) 再帰を用いた LU 分解（N=1000）．データサイズ $O(N^2)$, 計算量 $O(N^3)$.

測定環境・測定条件は以下の通りである．

- CPU
  - Pentium 4 3.00GHz
  - Xeon 3.20GHz
  - Core2 6400 2.13GHz
  - UltraSPARC-III 1.05GHz
- コンパイラ
  - XC-cube の GCC-3.4.6 ベース実装，`-O2 -march=pentium4`（Pentium 4, Xeon, Core2）`-mcpu=ultrasparc3`（UltraSPARC-III）
  - Intel C++ Compiler Version 10（UltraSPARC-III 以外かつ拡張仕様を含まない C コードのみ），`-O2 -march=pentium4`

表 2，**表 3** に性能測定の結果をまとめる．ここで，

表 3 性能測定 (生成・維持コスト) (2/2)
Table 3 Performance measurements (for creation and maintenance costs). (2/2).

S: UltraSPARCIII
P: Pentium4
X: Xeon
C: Core2Duo

Elapsed time in seconds (relative time to plain C)

		no closures	trampolines	closures	L-closures(SC)	L-closures
fib(37) load balancing	S(GCC)	0.888(1.00)	3.87(4.36)	1.33(1.50)	1.48(1.67)	1.18(1.33)
	P(GCC)	0.290(1.00)	0.524(1.81)	0.479(1.65)	0.631(2.18)	0.475(1.64)
	P(ICC)	0.290(0.999)	—	—	0.599(2.06)	—
	X(GCC)	0.318(1.00)	0.575(1.81)	0.541(1.70)	0.688(2.17)	0.525(1.65)
	X(ICC)	0.306(0.963)	—	—	0.643(2.02)	—
	C(GCC)	0.380(1.00)	0.554(1.46)	0.495(1.30)	0.738(1.94)	0.488(1.29)
	C(ICC)	0.306(0.805)	—	—	0.560(1.47)	—
Comp(20000) load balancing	S(GCC)	7.78(1.00)	22.0(2.83)	10.6(1.37)	8.76(1.13)	8.83(1.14)
	P(GCC)	2.84(1.00)	3.96(1.40)	3.34(1.18)	4.15(1.46)	3.46(1.22)
	P(ICC)	3.81(1.34)	—	—	4.93(1.74)	—
	X(GCC)	2.39(1.00)	3.96(1.66)	3.26(1.36)	4.09(1.71)	3.28(1.37)
	X(ICC)	3.41(1.43)	—	—	4.65(1.95)	—
	C(GCC)	2.50(1.00)	3.60(1.44)	3.18(1.27)	4.17(1.67)	3.52(1.41)
	C(ICC)	2.95(1.18)	—	—	3.50(1.40)	—
Pentomino load balancing	S(GCC)	2.73(1.00)	5.15(1.88)	4.66(1.70)	2.90(1.06)	2.75(1.01)
	P(GCC)	1.54(1.00)	2.19(1.42)	1.91(1.24)	1.98(1.29)	1.86(1.21)
	P(ICC)	1.50(0.975)	—	—	1.72(1.12)	—
	X(GCC)	1.09(1.00)	1.70(1.56)	1.52(1.39)	1.49(1.37)	1.37(1.26)
	X(ICC)	1.05(0.961)	—	—	1.39(1.28)	—
	C(GCC)	1.25(1.00)	1.77(1.41)	1.81(1.45)	1.53(1.22)	1.52(1.21)
	C(ICC)	1.12(0.894)	—	—	1.29(1.02)	—
LU(1000) load balancing	S(GCC)	3.06(1.00)	3.08(1.01)	3.05(0.995)	3.06(0.999)	3.08(1.01)
	P(GCC)	1.16(1.00)	1.16(1.000)	1.16(1.00)	1.16(1.00)	1.16(1.01)
	P(ICC)	0.719(0.621)	—	—	0.646(0.558)	—
	X(GCC)	1.05(1.00)	1.05(1.00)	1.05(1.00)	1.05(1.00)	1.05(1.00)
	X(ICC)	0.866(0.824)	—	—	0.768(0.732)	—
	C(GCC)	0.650(1.00)	0.650(1.00)	0.649(0.998)	0.651(1.00)	0.649(0.999)
	C(ICC)	0.457(0.703)	—	—	0.413(0.636)	—

表 4 性能測定 (呼び出しコスト)
Table 4 Performance measurements (for invocation costs).

		no closures	trampolines	closures	L-closures(SC)	L-closures
Quick sort	S(GCC)	0.417(1.00)	0.458(1.10)	0.433(1.04)	6.71(16.1)	5.33(12.8)
	P(GCC)	0.138(1.00)	0.487(3.54)	0.137(0.998)	1.88(13.6)	0.941(6.84)
	P(ICC)	0.128(0.932)	—	—	1.92(14.0)	—
	X(GCC)	0.145(1.00)	0.780(5.36)	0.129(0.887)	1.91(13.1)	0.881(6.06)
	X(ICC)	0.122(0.840)	—	—	2.00(13.8)	—
	C(GCC)	0.0390(1.00)	0.0388(0.993)	0.0394(1.01)	1.32(33.8)	0.644(16.5)
	C(ICC)	0.0366(0.939)	—	—	1.40(35.9)	—

“no closures” は高水準サービスを伴わない単なる C プログラムである．つまり，関数呼出し毎に入れ子関数定義をともなったり，入れ子関数のポインタを追加の引数として渡すことはしていない．“trampolines” は，GCC の従来の入れ子関数を使用した場合である．trampoline では，関数呼び出し頻度が高いなどのいくつかのプログラムでは倍近くまたはそれ以上の実行時間となっている．一方，L-closure はよい性能を示している．“no closures” と比較した相対実行時間が 1.00 にかなり近いものが多く見られる．

また，“L-closures(SC)” は変換に基づく LW-SC 処理系により実装された L-closure であり，GCC ベース実装の L-closure に準ずる性能が得られている．また L-closure(SC) では，出力 C コードを GCC よりも最適化が期待できる Intel C++ コンパイラ（表では ICC）でコンパイルできる．例えば，BinTree, Bin2List や Pentomino などに見るように，ICC を使った L-closure(SC) のほうが，GCC ベース実装の L-closure よりも性能がよいこともある．このことから，理論的には ICC に対して L-closure を導入でき

ればもっとよい結果が得られると予想される．

ここで，表 2，表 3 をみると，IA-32 上の負荷分散を伴う fib(37) と nqueens(13) については，やや例外的に L-closure の性能が悪かった．また，Comp(20000) については，L-closure の性能が closure よりも劣っていた．これらでは，重要ではない変数にレジスタが割り当てられていた．IA-32 は少数の callee セーブレジスタしか持たず，また callee セーブレジスタの保存/復元は明示的に実行する必要があるため，間違った割り当てのペナルティは相対的に重くなる．我々の私用場所を足すという手法は，レジスタ割り当ての候補を増やす効果は持っているが，よい割り当ての機会だけでなく悪い割り当ての機会も増やすことになる．一方，より多くの callee セーブレジスタを持ち，レジスタウィンドウによってその保存/復元が遅延されている SPARC においては，L-closure あるいは L-closure(SC) という我々の手法はきわめて安定して効果的であることがわかる．

また，L-closure の呼出しコストは高くなっても構わないものとしたが，実際にどの程度の呼び出しコストなのかを，クイックソートの比較器として通常の関数ポインタまたは closure ポインタを渡してクイックソート本体から間接的に呼び出してもらうという評価を行った．もちろん，これは高水準サービス実装ではない．**表 4** に示すとおり，L-closure や L-closure(SC) では呼び出しコストが高いことがわかる．これは意図通りであり，そのような呼び出しが頻繁な場合には XC-cube の closure を用いればよいことが確認できる．IA-32 の trampoline での呼び出しコストの高さについては，スタックというよく書き込む領域にコードを置くことで性能が落ちていたのではないかと考えていたが，Core2 ではこの速度低下は見られなかった．

8. 高水準サービス（関連研究）

C コンパイラをベースとしてごみ集めなどの高水準サービスを実装する方法は少なくとも次の 4 つのカテゴリに分類することができる．

(1) C 言語自身で直接的スタック操作を行う[1]．正当性や移植性に疑問．
(2) 特殊なサービスルーチン群をアセンブリレベルで提供し，C への翻訳系においてそのルーチン群を利用する[21]．
(3) 手の込んだ翻訳技法により，C 言語へと翻訳する[4),6),7),15),18),22),24]．
(4) C コンパイラを拡張し，新たに備えた機能を使って拡張 C 言語へと翻訳する．[5),20]．

LW-SC[9),10] は 3 番目，本論文で主題としているコンパイラベースの実装[25] は，4 番目の方式をとっている．

以下では，1 章，2 章で注目したごみ集め以外の高水準サービスについて，入れ子関数の形でプログラム中に記述される L-closure がどのように貢献できるかを関連研究と比較しながら議論する．

8.1 スタックの状態のキャプチャと復元

入れ子関数を使えば，呼び出し元に戻ることなくスタックの状態をキャプチャすることができる．付録 A.1 はキャプチャするための入れ子関数を持つ関数を示している．このプログラムでは，現在の実行点を表す擬似プログラムカウンタ pc の値を更新しながら通常の計算を進めておき，キャプチャするときには，pc に加えてすべてのパラメータや局所変数の値をセーブする．また，呼び出し元のセーブ用入れ子関数を呼び出すことで，スタック全体について状態のキャプチャができる．なお，すでにキャプチャした状態を復元するのはずっと簡単であり，入れ子関数は必要としない．また効率の面では復元用には別バージョンの関数を準備することも考えられる．この技法は，チェックポインティング，マイグレーション，一級継続などの高水準サービスの実現に応用できる．

C 言語への手の込んだ翻訳技法によりチェックポインティングを実現したものには Porch[18] がある．スタックの状態の保存時には，プログラムは擬似プログラムカウンタ（call-id），パラメータ，局所変数の値を保存しながら，スタックの底に達するまで順次リターンしていく．復元時にはこのプロセスを反転させる．キャプチャの際にリターンしなくてはならないのは困るが，チェックポイントをとった結果を用いれば実行が再開できる．LW-SC[9),10] の変換方式もリターン後に再開するために同様の手法を用いている．LW-SC の場合は保存した状態を表す外部スタックに入れ子関数がアクセスすることで状態の参照・更新が可能となっていると考えることもできる．

8.2 マルチスレッド（遅延隠蔽）

各関数に自分と同等の計算を続けるための入れ子関数を持たせ，その入れ子関数のポインタを保存してスレッドの未処理の計算（継続）を先行して実行したい場合に呼び出せるようにすることによって高水準言語レベルの遅延隠蔽のためのマルチスレッドを実現することができる[8),19]．つまり，通常の暗黙の継続（　スタック）以外に，明示的な継続を入れ子関数により表現し，継続渡しスタイルの要領で呼び出し先に渡しておく．普段は，暗黙の継続のほうを用いて普通にリターンすることで明示的な継続は使われないが，スレッド

A が実行を続けられない場合にもスタック中で眠っている他のスレッド B の明示的継続を使えばスレッド B の処理を先に進めることができる[19].

遅延隠蔽を主な目的としたマルチスレッドのためにコンパイラを拡張した実装には，StackThreads/MP[20]⋆1や Lazy Threads[5] が挙げられる．ともに，現在のスレッド A をサスペンドさせて親スレッド B に切り替えるメカニズムを提供しており，B を動作させるために，フレームポインタを，スタックトップ用のポインタとは独立に B に合わせられるようになっている．両者には，この状態で B が新たにフレームを割り当てるときにスタックトップ用のポインタから割り当てるか，別にスタックレット（少量のスタック）を確保して割り当てるかの差がある．これに対し，我々の提案する L-closure は後述の負荷分散を目的とする場合を含めマルチスレッド専用ではなく，ごみ集めなど様々な用途に使えるようになっている．

またライブラリ実装による StackThreads[21] では，子スレッドのコンテクストを保存して親スレッドに切り替える `switch_to_parent` と，コンテキストを復元して実行再開するための `restart_thread` という 2 つの特殊なルーチンを使い，子スレッドのフレームを邪魔にならないところに保管しておくことでマルチスレッドを実現する．これらのルーチンは callee セーブレジスタの扱いに注意してアセンブリレベルで実装されている．また，このように子スレッドのフレームを邪魔にならないところに移動させることに相当する処理を，C 言語への手の込んだ翻訳技法により実現したものには，Concert[15] や OPA[22],[24] がある．どちらも，子スレッドを通常の関数呼び出しの形で実行しておき，サスペンドする場合には子スレッド自体が自分の状態をヒープから割り当てる明示的フレームに保存する．この保存，または，後での実行再開するとき⋆2の復元は C 言語の構造体や局所変数などにアクセスする形の正式なものとなっている．これら，フレームの移動を伴う方式と比較して，入れ子関数による実装には，サスペンド時のフレームの移動が不要という点と，アドレスがとられるデータをフレーム中に持てるという点がある．

8.3 負荷分散

探索問題などの樹状再帰的な不均質な細粒度並列計算においてプロセッサ間でタスクを授受して効率よく負荷分散を行うには，実行中のタスクから lazy に並列実行可能なタスクを分割・生成し抜き取れるようにした遅延タスク生成[12]（Lazy Task Creation; LTC）などの負荷分散方式が有効である．もともとの LTC[12] では，論理的なスレッド生成を意味する `future` 構文の式の実行において，新しいスレッドによる `future` の対象となる式の評価実行をそのまま行い，元のスレッドの `future` の後の**継続**はスタックに保存しておく．他の idle なプロセッサがそのようなスレッドの継続のうち，最も古いものをアセンブリレベルでタスクとして盗むこともできるようにすることで負荷分散を行う．通常，thief プロセッサは犠牲者となるプロセッサをランダムに選択する．また，メッセージパッシング型遅延タスク生成[3] では，idle なプロセッサは継続（タスク）を直接盗むのではなく，まずタスクの要求を行い，要求されたプロセッサが，自分で継続からタスクを生成して要求したプロセッサに渡す．メッセージパッシング型遅延タスク生成に基づき，先に述べた StackThreads/MP[20] や Lazy Threads[5] も負荷分散を実現している．

入れ子関数の形で L-closure を使うと，LTC の本質的な部分に基づく，低オーバヘッドの負荷分散フレームワークを構築できる[13],[23],[26]．入れ子関数により，呼び出し元の変数にアクセスできるため，スタックの底のほうであっても，タスク生成に必要な情報を取り出したり，取り出したことを記録したりできるためである．この遅延分割型負荷分散フレームワークにおいては，タスク要求があるまで論理的なスレッド生成もせずに実行を進める．タスク要求がくると，一時的ではあるが，できるだけ実行をバックトラックさせ，初期段階（スタックの底付近）で生成可能だったが当初生成を見送ったスレッドをできるだけ大きなタスクとして生成する．その後，すでに済んだ計算はできるだけ省略してタスク要求を受けた時点から実行を再開する．入れ子関数を用いれば，一時的にバックトラックする際に，そのまでの副作用を一時的に取り消して，スレッド/タスクを生成後，再び副作用を起こすことも可能である．これは，その場更新により逐次処理では避けることのできるバックトラック探索上一段進めた部分解（partial solution）の毎回のコピーを，複数プロセッサを用いた遅延分割型負荷分散でも，比較的簡単なバックトラック記述で避けられるという点において効果的である[23],[26]．また，関連研究の多くがマルチスレッド間の共有メモリを想定しているのに対し，分割時に送受信を伴わせることで分散環境でも動作可能となる[11],[13].

C 言語への手の込んだ翻訳技法により LTC ベース

⋆1 GCC 出力アセンブリコードの後処理による実装

⋆2 再開には別バージョンの関数を用いる．

の負荷分散を実現するものには，高水準マルチスレッド言語の Cilk[4] や OPA[22),24] などがある．Cilk や OPA の翻訳系は 2 バージョン（fast/slow）の C コードを生成する．fast バージョンは Cilk の場合は呼び出し時に，OPA の場合はサスペンドによるリターン時に生きている変数の値をヒープ割り当てのフレームに保存し，slow バージョンがヒープ上のフレームが表す継続に基づいて残りの計算を続られるようにする．OPA ではメッセージパッシング型 LTC としてリターン時まで保存を遅らせられるが，Cilk では直接盗めるようにするため呼び出し時に保存を行う．

9. 議　　論

特殊なコンパイラでスタック上のルート変数マップなどのアノテーションを生成することでごみ集め処理などの高水準サービスを実現する方式と比較すると，「構造体とポインタ」に基づく翻訳技法[6),7] などと同様に，XC-cube の closure や L-closure のコンパイラ実装は，稀にしか実行されないコードがアノテーション量よりも通常大きなコードスペースを消費する．改善案としては，稀にしか実行されないコード（入れ子関数のコードのほか，L-closure の場合は前処理などのコード）を別のコードセグメントにまとめることが考えられる．これにより普段実行されるコードのための命令キャッシュや命令 TLB（または物理メモリ）を汚すことはなくなるので，実行性能面でのペナルティをなくすことはできる．

図 2 のプログラムで n フレームの実行スタックをスキャンするときには，4 章の実装モデルでは L-closure のネストした呼び出しのために n フレームが追加で必要となる．このスタックの消費が倍になるという点が問題であれば，次に呼び出してほしい関数を呼び出し元にリターンすることで末尾呼び出しを除去するというよくある技法を用いることは可能である．一方，時間計算量についていうと，このプログラムでの「1 つ前の呼び出し元への一時的リターン」の回数は $O(n^2)$ となっている．この点が問題であれば，以下に述べるもう一つの L-closure の実装モデルを採用すべきである．それは，L-closure を呼び出すときには，スタック全体について，前処理が済んでいない呼び出しインスタンス（フレーム）が無いように一括して変換するというものである．このとき変換するのは，最後に L-closure を呼び出してから今までの間に新しく作られた呼び出しインスタンスに関してのみでよい．最初にすべて変換する場合も「1 つ前の呼び出し元への一時的リターン」の回数は $O(n)$ であり，その後，差分のみ変換するときにはもっと少ない回数で済む．少なくとも性能評価や関連研究でとりあげた高水準サービスについては，マルチスレッドを除いて，n フレーム全体について L-closure を呼び出すことになるものであり，将来的にはこの実装モデルを採用すべきと考えている．

性能測定の結果からは，呼び出し先に L-closure のアドレスを常に追加の引数として渡すというコストは問題にならないものであった．このコストも削減したい場合は，例外ハンドラを探す場合と同様に L-closure を安全に探すためのタグ（引数の型をタグとしてもよい）を決めておき，直接 L-closure のアドレスを指定する代わりにタグを指定して呼び出しをするという方法が考えられる．もともと，L-closure のコンセプトとしては，例外ハンドラと同じように稀にしか使われないということを想定しており，例外ハンドラの場合は例外ハンドラまで非局所脱出してその実行後に元の計算が続けられないのと比較して，L-closure の場合は，その呼び出しの後，元の計算が続けられる．

10. おわりに

本研究では，合法的実行スタックアクセスのための新しい言語機構 “L-closure” 提案している．L-closure を使って，高性能・高信頼プログラミング言語が備えるごみ集めなどの様々な高水準実行時サービスをすっきりと効率よく実装できる．プログラムの通常実行時のオーバヘッドを 0 に近づけるため，L-closure の性能目標は，呼び出しコストは高くても構わないので，生成・維持コストをできるだけ削減するというものとした．既存の GCC に基づき L-closure を実装する方法として，既存の最適化器などをそのまま利用できる方法の詳細を示すともに，少ないコード量で実装可能であることを示した．

また，性能測定の結果，意図したとおりのきわめて小さい L-closure の生成・維持コストにより，頻繁に L-closure を生成しつつ稀にしか呼び出さない多くの高水準サービスにおいて，トータルのオーバヘッドが大きく削減可能なことを示した．

L-closure は，例外ハンドラと似て，設定コストを抑えつついざというときは使える機能を提供する．典型的な例外ハンドラと異なるのは，利用時に，例外ハンドラまでの非局所脱出がなされて元の計算が続けられなくなることないという点である．これにより，高性能・高信頼プログラミング言語の実装向け言語機構としての価値が高いと考えている．

L-closure のコンパイラ実装は，実行性能がよくて

も距離をおきたいという印象を持たれるかもしれない．その意味で，C 言語への変換によってこれに準ずる性能を達成できる LW-SC 処理系は，新しいターゲットマシンのサポートが容易という点を含めて，L-closure のコンセプトへの共感を得るのに必要なものと考えている．また性能を重視しない場合は，GCC の入れ子関数をそのまま使うことで応用事例は共有できる．

今後の課題には， L-closure の機能を持つ拡張 C 言語 XC-cube を中間言語として実際に利用して，様々な高水準言語を実装することが挙げられる．また，高水準サービスのうちごみ集めについては，様々な高水準言語で共通であることが多いため，そのような共通中間言語が設計できないかを模索している．また真の末尾再帰をサポートすべき高水準言語も多いためそのサポートも重要な課題である．

現在，GCC ベース実装は 32bit 版の SPARC と IA-32 をサポートしているが，他のターゲットマシンもサポートしていきたい．特に，x86-64 のサポートは重要であると考えている．また，IA-32 よりレジスタ数が多い SPARC のほうが L-closure の効果が顕著なことから，IA-32 からレジスタ数が増えている x86-64 でも高い効果が期待できる．

謝辞 本研究の一部は，文部科学省科学研究費萌芽研究 17650008 の補助を得て行った．

参 考 文 献

1) Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice & Experience*, Vol.18, No.9, pp. 807–820 (1988).
2) Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
3) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proceedings of International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No.748, Springer-Verlag, pp.94–107 (1993).
4) Frigo, M., Leiserson, C. E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI'98)*, Vol.33, No.5, pp.212–223 (1998).
5) Goldstein, S.C., Schauser, K.E. and Culler, D.E.: Lazy Threads: Implementing a Fast Parallel Call, *Journal of Parallel and Distributed Computing*, Vol.3, No.1, pp.5–20 (1996).
6) Hanson, D. R. and Raghavachari, M.: A Machine-Independent Debugger, *Software – Practice & Experience*, Vol.26, No.11, pp.1277–1299 (1996).
7) Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. of the 3rd International Symposium on Memory Management*, pp.150–156 (2002).
8) 平石　拓，李　暁ろ，八杉昌宏，馬谷誠二，湯淺太一：S 式ベース C 言語における変形規則による言語拡張機構，情報処理学会論文誌：プログラミング， Vol.46, No.SIG1 (PRO24), pp.40–56 (2005).
9) Hiraishi, T., Yasugi, M. and Yuasa, T.: Implementing S-Expression Based Extended Languages in Lisp, *Proceedings of the International Lisp Conference*, pp.179–188 (2005).
10) Hiraishi, T., Yasugi, M. and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol. 2, pp. 262–279 (2006). (IPSJ Transactions on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50–67).
11) Hiraishi, T., Yasugi, M. and Yuasa, T.: Experience with SC: Transformation-based Implementation of Various Language Extensions to C, *Proceedings of the International Lisp Conference*, pp.103–113 (2007).
12) 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).
13) 大林竜太，平石　拓，八杉昌宏，馬谷誠二，湯淺太一：遅延分割型負荷分散フレームワークの試験実装，情報処理学会第 55 回プログラミング研究会 (SWoPP2005) (2005).
14) Peyton Jones, S., Ramsey, N. and Reig, F.: C--: a Portable Assembly Language That Supports Garbage Collection, *International Conference on Principles and Practice of Declarative Programming* (1999).
15) Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A. A.: A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers, *Supercomputing'95* (1995).
16) Ramsey, N. and Jones, S.P.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions, *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.285–298 (2000).
17) Stallman, R.M. and the GCC Developer Community: *GNU Compiler Collection Internals*, Free Software Foundation, Inc. (2004).
18) Strumpen, V.: Compiler Technology for

Portable Checkpoints, http://theory.lcs.mit.edu/~strumpen/porch.ps.gz (1998).

19) 田畑悠介，八杉昌宏，小宮常康，湯淺太一：入れ子関数を利用したマルチスレッドの実現，情報処理学会論文誌：プログラミング, Vol.43, No.SIG3 (PRO14), pp.26–40 (2002).

20) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pp. 60–71 (1999).

21) Taura, K. and Yonezawa, A.: Fine-grain Multithreading with Minimal Compiler Support – A Cost Effective Approach to Implementing Efficient Multithreading Languages, *Proc. of Conference on Programming Language Design and Implementation*, pp.320–333 (1997).

22) Umatani, S., Yasugi, M., Komiya, T. and Yuasa, T.: Pursuing Laziness for Efficient Implementation of Modern Multithreaded Languages, *Proc. of the 5th International Symposium on High Performance Computing*, Lecture Notes in Computer Science, No.2858, pp.174–188 (2003).

23) 八杉昌宏，小宮常康，湯淺太一：入れ子関数を利用する動的負荷分散と高水準記述，情報処理学会論文誌：コンピューティングシステム, Vol.45, No.SIG11 (ACS7), pp.368–377 (2004).

24) 八杉昌宏，馬谷誠二，鎌田十三郎，田畑悠介，伊藤智一，小宮常康，湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法，情報処理学会論文誌：プログラミング, Vol.42, No.SIG11 (PRO12), pp.1–13 (2001).

25) Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, pp. 170–184 (2006).

26) Yasugi, M., Komiya, T. and Yuasa, T.: An Efficient Load-Balancing Framework Based on Lazy Partitioning of Sequential Programs, *Proceedings of Workshop on New Approaches to Software Construction*, pp.65–84 (2004).

27) 湯淺太一，中川雄一郎，小宮常康，八杉昌宏：リターン・バリア，情報処理学会論文誌：プログラミング, Vol.41, No.SIG9 (PRO8), pp.87–99 (2000).

付　　録

A.1 Capturing State with L-closures

```
int cpfib(void (*save0) lightweight (), int n)
{
  int pc = 0;      /* pseudo program counter */
  int s = 0;
  void save1 lightweight (){    /* L-closure */
    save0();          /* save caller's state  */
    save_pc(pc);      /* save pc state        */
    save_int(n);      /* save variable state  */
    save_int(s);      /* save variable state  */
  }
  if (n <= 2) return 1;
  pc = 1; /* inc program counter before call */
  s += cpfib(save1, n-1);
  pc = 2; /* inc program counter before call */
  s += cpfib(save1, n-2);
  return s;
}
```

A.2 Compilation Output for Appendix A.1

```
 1  cpfib:
 2          !#PROLOGUE# 0
 3          save    %sp, -160, %sp
 4          !#PROLOGUE# 1
 5          add     %fp, -32, %l1
 6          mov     %i0, %l0
 7          add     %i1, -1, %o1
 8          mov     %l1, %o0
 9          cmp     %i1, 2
10          mov     1, %i0
11          ble     .LL1
12          mov     0, %l2
13          call    cpfib, 0
14           nop
15  .LL17:
16          ! pjmp  .LL16
17  .LL13:
18          add     %l2, %o0, %l2
19          add     %i1, -2, %o1
20          mov     %l1, %o0
21          call    cpfib, 0
22          mov     2, %i0
23  .LL18:
24          b       .LL11
25           nop
26  .LL19:
27          cmp     %o0, 0
28          be,a    .LL10
29          st      %l2, [%fp-24]
30          b       .LL14
31          ld      [%fp-24], %l2
32  .LL11:
33          b       .LL1
34          add     %l2, %o0, %i0
35  .LL16:
36          cmp     %o0, 0
37          be,a    .LL12
38          st      %i1, [%fp+72]
39          ld      [%fp-24], %l2
40          ld      [%fp+72], %i1
41          ld      [%fp+68], %l0
42          ! pjmp  .LL13
43  .LL14:
44          call    __cont_return, 0
45          add     %fp, -48, %g2
46  .LL12:
47          st      %i0, [%fp-20]
48          st      %l0, [%fp+68]
49          st      %g0, [%fp-24]
```

```
50  .LL8:
51          sethi   %hi(save1.0), %g1
52          add     %fp, -16, %o5
53          st      %o5, [%fp-28]
54          or      %g1, %lo(save1.0), %g1
55          add     %fp, -48, %g2
56          st      %g1, [%fp-32]
57          call    __hook_and_cont_L_call, 0
58          nop
59          nop
60          std     %o0, [%fp-64]
61          std     %f0, [%fp-56]
62          add     %fp, -48, %g2
63          add     %fp, -64, %g1
64          call    __cont_post, 0
65          st      %g1, [%fp-44]
66  .LL10:
67          st      %i1, [%fp+72]
68          st      %i0, [%fp-20]
69          b       .LL8
70          st      %l0, [%fp+68]
71  .LL1:
72          ret
73          restore
74          unimp 1; unimp 1
75  .LL2:
76          cmp     %g2, %fp
77          bg      .LL20
78          nop
79          call    __dup_cont
80          nop
81          cmp     %o1, .LL2-(.LL18-8)
82          be      .LL19
83          cmp     %o1, .LL2-(.LL17-8)
84          be      .LL16
85          nop
86          call    __cont_L_call
87          nop
88  .LL20:
89          call    __stack_walk
90          nop
```