Vol.2011-OS-118 No.22
2011/7/28

LLVM を用いたオブジェクトファイルの細分化

河 合 夏 輝[†1]　笹 田 耕 一[†1]

本稿では，ソースコードをコンパイル，リンクした際に得られる実行ファイルや共有ライブラリといったオブジェクトファイルを，小さな起動プログラムと多数の共有ライブラリに分割する機構について，その手法と設計を述べる．一般に，プログラムを実行する際には，ストレージなどに置かれた実行ファイルをページ単位でロードする．これに対して，我々の分割機構を用いてプログラムを関数単位で分割することにより，プログラムのロードをより柔軟に制御し，効率的に行うことができる．分割機構は，オブジェクトファイルに含まれる関数を最小単位として，1 つの関数を 1 つの共有ライブラリに分割する．そして，生成した共有ライブラリを必要に応じてロードしながら実行する起動プログラムを生成する．また，関数単位に分割すると，総ファイルサイズが増加するため，適切に関数群を設定し，1 つの共有ライブラリとしてまとめる．プログラムの細分化には，様々なプログラムを対象とするため，広く利用されている LLVM を利用する．さらに，分割機構の応用例として，プログラムを狭帯域なネットワークから効率よく起動する例を紹介し，実際に既存のプログラムを細分化した結果について述べる．

Scattaring Object files by LLVM

Natsuki Kawai[†1] and Koichi Sasada[†1]

This paper describes the system scatters executable files or shared libraries, generated by compile and link processes, into a small launch program and numbers of shared libraries. When we execute a program, we load executable file from a storage to a memory. On the other hand, we can controll the loading process of a scattered program more flexibly and effectively. Our system assumes that a function is a smallest part of an object file and scatters an object file into libraries each of them contains one function. Our system also generates a launch program loads and executes the scattered shared libraries. To avoid a size-overhead of each shared library, our system generates function groups. We implemented our system on LLVM, a commonly used compiler infrastructure, to target many programs. As one of the applications of this system, this paper describes a way to run programs over a narrow network. Moreover, this paper evaluates file sizes and performance of programs scatterd by our system.

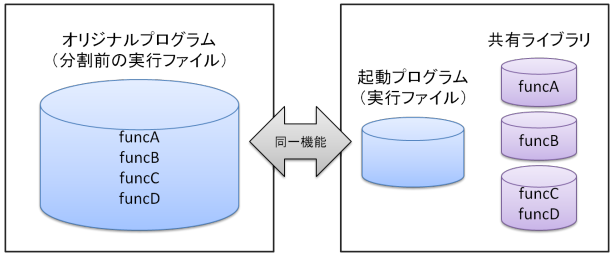

図 1　プログラムの分割
Fig. 1　Scattering a Program

1. はじめに

プログラムを実行する場合，メモリ管理を行う通常の OS では，プログラム全体を仮想メモリ空間にマップし，物理メモリへのロード，アンロードなどの処理をページ単位で行う．そのため，ページ単位でロードする場合は，実際に実行されない個所も余計にロードされてしまう可能性がある．最近の高性能な計算機では，このオーバヘッドが問題となることは少ないが，例えば組み込み機器など，動作速度がまだ十分でない計算機では無視できない負荷となり得る．また，インターネットなど，ネットワーク上にあるプログラムを実行する際，多くの場合，プログラムをすべてダウンロードしなければ実行することができない．必要な機能ごとにダウンロードするように変更する場合は，手作業でプログラムの分割を行う必要がある．

そこで，我々は LLVM[1)] を用いてプログラムを自動的に分割する機構を開発した．本分割機構は，プログラムを起動プログラムといくつかの共有ライブラリに細分化する．起動プログラムは，必要に応じて共有ライブラリを読み込み，オリジナルのプログラムと全く同じ動作を行う．

分割機構では，一般的な実行ファイルや共有ライブラリファイルをコンパイルする際に，関数[⋆1]をプログラムの最小構成要素として捉え，1 つの関数につき，1 つの共有ライブラリ

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

⋆1 本稿では，C 言語における関数を，単に関数と呼ぶ．戻りアドレスをスタック上に配置した状態で呼び出され，

を生成する．ただし，1 関数 1 共有ライブラリという細分化では，総ファイルサイズが増大するという問題があるため，分割された共有ライブラリ同士を，適切なグループに分けて再度結合する（**図 1**）．

本稿の構成は次の通りである．2 章で分割機構の全体像を述べ，起動プログラムがどのように元のプログラムと同様の機能を果たすかについて述べる．3 章では，既存のプログラムを起動プログラムと，関数 1 つにつき 1 つの共有ライブラリに分割する方法について示し，4 章で，LLVM での実装について述べる．5 章では，関数をグルーピングし，複数の関数を一つの共有ライブラリに収める手法を提案する．6 章では応用例として，プログラムを自動的に分割し，ネットワーク上から段階的にダウンロードして実行する例を紹介する．7 章では，実際に既存のプログラムを分割し，提案手法及び開発した機構の性能評価を行う．そして，8 章では関連研究について述べ，9 章では本稿のまとめと今後の課題，展望を示す．

2. 分割機構の全体像

我々は，対象とするプログラムを起動プログラムといくつかの共有ライブラリに分割することで，ロード処理をより細かな単位で制御することを提案する．

この起動プログラムは，分割された共有ライブラリを適切にロード，実行することで，オリジナルプログラムと同じ動作をする．本章では，まず本機構をどのように利用するかを 2.1 節で述べ，起動プログラムと共有ライブラリが果たす役割を，それぞれ 2.2 節，2.3 節で述べる．

2.1 分割機構の利用

本分割機構は，元のプログラムを C のソースコードという形で入力として受け付け，出力として起動プログラム，および細分化した複数の共有ライブラリを生成する（**図 2**）．本分割機構の利用者は，生成された起動プログラムを実行することで，元のプログラムと同等の結果を得ることができる．

起動プログラムは，生成された共有ライブラリを必要に応じてロードしながら実行していく．そのため，利用されなかった機能を含む共有ライブラリはロードされない．

2.2 共有ライブラリ

共有ライブラリは，オリジナルプログラムに含まれていたプログラムの実体（テキスト領域）を，まず関数単位で分割し，次に適切な組み合わせで統合し，一つのファイルに収めたものである．共有ライブラリは，一つのオリジナルプログラムに対して複数生成される．また，オリジナルプログラム中のデータ領域のうち，一つの関数からしか直接アクセスされないデータもこのファイルに含める．起動プログラムは，このライブラリを動的にロードし，収められている関数のアドレスを取得し，実行する．

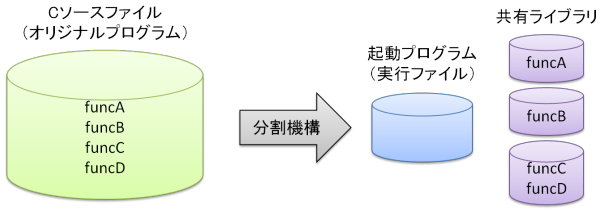

図 2 分割機構の動作
Fig. 2 Functionality of the Scattering System

2.3 起動プログラム

起動プログラムは，前節で述べた共有ライブラリを適切にロードすることでオリジナルプログラムと同じ動作を行うプログラムである．利用者は，オリジナルプログラムの代わりに，この起動プログラムを実行する．

分割機構は，起動プログラム上に，分離された関数 1 つにつき，対応する関数（スタブ）を用意する．スタブは，対応する関数が格納されている共有ライブラリがロードされていない場合，それをロードし，実行する．すでにロード済みであれば，そのまま関数呼び出しを行う．関数を呼び出す側から見ると，(実行時間を除けば）オリジナルプログラムにおける関数呼び出しとまったく同様と見ることができる．

起動プログラムには，共有ライブラリを管理する共有ライブラリマネージャ（マネージャ）を用意する．マネージャはロードした共有ライブラリを記憶し，同じ共有ライブラリの重複ロードを回避する．また，ロード済みの関数群を管理し，要求された関数を呼び出すことができる．

この他に，複数の関数から参照されるデータ領域についても，起動プログラムに配置する．これにより，複数の共有ライブラリから同一のデータへアクセスすることが可能となる．

スタブとマネージャの具体的な実装，動作については，3.3，3.4 節にて述べる．

実行を終えるとその戻りアドレスに制御を渡すものを指す．

3. 分割手法

本章では，プログラムを 1 関数につき 1 つの共有ライブラリへと分割する手法について示す．

本分割機構は，プログラムに含まれる関数を最小単位として捉え，プログラムを関数単位に分割する．分割した関数は，共有ライブラリとしてそれぞれ生成する．なお，現実的には，関数 1 つにつき 1 つの共有ライブラリとすると，総ファイルサイズが増大してしまうので，最終的にはいくつかの関数を 1 つの共有ライブラリにまとめるが，これについての詳細は 5 章で述べる．

共有ライブラリへの分割は，オリジナルプログラムのテキスト領域と一部データ領域を関数単位で共有ライブラリとして切り出し，独立したファイルを生成することで行う．切り出した関数を呼び出す側は，切り出した関数に相当するスタブを呼び出す．

スタブを埋め込んだファイルを関数 ID に基づいて共有ライブラリの管理を行う共有ライブラリマネージャと共にリンクすることで起動プログラム生成する．

以上の手順のうち，共有ライブラリとしてテキスト領域と一部データ領域を切り出す手順を 3.1 節で示し，データ領域のうち共有ライブラリに切り出していない部分の取り扱いについて 3.2 節で取り上げる．スタブの生成方法と機能を 3.3 節で述べる．最後に，マネージャの機能について 3.4 節で示す．

3.1 共有ライブラリの切り出し

オリジナルプログラムのテキスト領域とデータ領域の一部を切り出し，オリジナルプログラム中の 1 関数に対して 1 つの共有ライブラリを生成する（**図 3**）．

共有ライブラリをロードし，関数を取り出す際には，関数の識別子が必要となるが，オリジナルプログラムにおける関数名を識別子として使用した場合，その管理のために多くのメモリが必要となってしまう．そこで，関数を共有ライブラリとして切り出す際には，分割の対象となる全ての関数に，その関数名の代わりに，プログラム全体で一意となる関数 ID を割り当てる[⋆1]．次に，起動プログラムから共有ライブラリ中の関数にアクセスすることができるよう，この関数名シンボルを公開する．これにより，マネージャが関数 ID から関数名を導き，関数のアドレスをひくことができるようになる．さらに，オリジナルプログラム中のデータ領域のうち，この関数から参照されていない変数を関数と共に切り出す．この時，

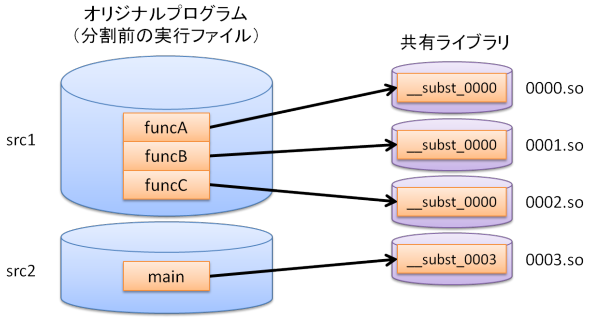

図 3 共有ライブラリの切り出し
Fig. 3 Generating a Shared Library

ファイルの内部からは参照されていなくても，シンボルを外部に公開しているならば他の関数から参照されているものとみなし，切り出さない．

3.2 データシンボルの公開

前節では，データ領域のうち，直接アクセスする関数が高々一つのものについては，アクセスする関数とともに共有ライブラリに切り出すことを述べた．対して，複数の関数からアクセスされるデータについては，起動プログラム中に配置する．しかし，切り出した共有ライブラリからは起動プログラムが公開していないデータにアクセスすることはできないが，単純にシンボルを公開した場合，その分起動プログラムのサイズが増加する．そこで，オリジナルプログラムに残っている非公開シンボルのうち，データ領域を参照するものについては，インデックスとオフセットの形で公開する．

3.3 スタブの生成と動作

関数切り出しの後，切り出した関数に代わるスタブを生成する．スタブは元の関数の代わりとなる関数なので，元の関数のシンボルが外部に公開されている場合，スタブにも外部からアクセスできるようにスタブには元の関数と同じ名前をつける．また，元の関数シンボルを公開していない場合でも，共有ライブラリからスタブにアクセスする可能性があるので，そのシンボルを公開する必要がある．この場合，関数名の衝突を避けるため，プログラム全体で一意に定まる名前を付与する．プログラムによっては，多くの関数名を公開することに

⋆1 我々の実装では，“__subst_(関数 ID の 16 進表現)” とした．

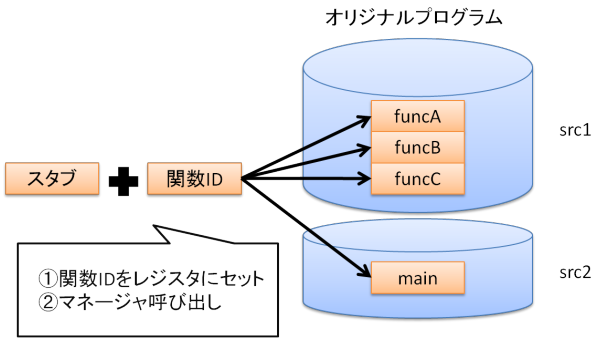

図 4 スタブの埋め込み
Fig. 4 Embed Stubs

なるので，この名前は極力短いものとする．[*1]（**図 4**）．

次に，スタブの持つ機能について述べる．スタブは関数ごとに生成されるものなので，そのサイズが大きいと起動プログラムのサイズが大きく増加する．そこで，我々は，スタブには最低限の機能のみを持たせ，残りの処理を後述の共有ライブラリマネージャに移譲した．前述の通り，分割機構は，関数分割の際に関数を一意に識別可能な ID を付与している．従って，マネージャに関数 ID を渡すことができれば，マネージャは適切に関数を選択，実行することができる．そこで，我々はスタブに，関数 ID のマネージャへの受け渡しと，マネージャへの処理の移譲のみを実装した．これらの処理は，レジスタに即値を設定する命令と即値へジャンプする命令を持つ CPU 上で，関数 ID 渡しに使う空きレジスタを確保できる環境であれば，2 命令で実装することができる．例えば，我々は，x86 命令セットと cdecl 呼び出し規約を組み合わせた環境において，movl 命令と jmp 命令の 2 命令，10byte で実装した（**図 5**）．詳細は次節で述べるが，この実装ではスタブはスタックフレームを構築しないので，呼び出し元が積んだ引数をそのまま共有ライブラリ上の関数に渡すことも容易となる．

3.4 共有ライブラリマネージャ

共有ライブラリマネージャは，関数 ID をレジスタから受け取って，該当する関数を適切

```
movl $1  %eax
jmp ⟨shared-lib manager⟩
```

図 5 x86+cdecl 環境におけるスタブ
Fig. 5 A Stub under x86+cdecl Environment

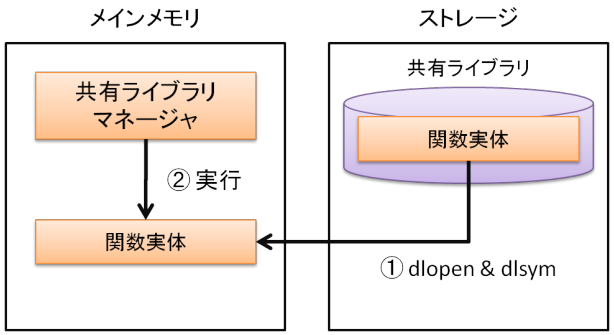

図 6 共有ライブラリマネージャ
Fig. 6 Shared Library Manager

にロード，実行する機構である（**図 6**）．共有ライブラリマネージャは，共有ライブラリへ分割された関数が呼び出されたタイミングで実行される．内部では任意の処理を行うよう，カスタマイズすることができる[*2]が，本節では共有ライブラリマネージャが行う必要のある最低限の処理について述べる．

マネージャは，関数 ID から関数アドレスをひくためのテーブル（以降，関数テーブルと呼ぶ）を持つ．また，マネージャには，C 言語の main 関数など，エントリポイントとなる関数のスタブから呼び出されるものと，それ以外のスタブから呼び出されるものの 2 種類が存在する．前者は，関数実体のロード，呼び出しの他，関数テーブルの初期化処理を行う．初期化処理では，分割された関数全てのポインタを収めるための領域を関数テーブルとして確保し，このテーブル全体を，無効なアドレス値（NULL）で初期化する．

次に，関数のロード処理について述べる．スタブから呼び出されたマネージャは，まず，引数である関数 ID を用いて関数テーブルをひく[*3]．ここで有効なアドレスが得られた場合，

⋆1 我々の実装では，“$(関数 ID の 16 進表現)” とした．

⋆2 6 章では，このカスタマイズ機構を用いた例を示す
⋆3 関数引数のレジスタ渡しが行われる環境では，この時点で引数渡しに使われるレジスタ値をスタック上に保存す

ID に対応する関数アドレスとしてこの値を使用する．NULL であった場合，ID に対応する関数は未ロードであると判断し，共有ライブラリの動的ロードを行う．動的ロードは，共有ライブラリのファイル名を関数 ID から導出し，これを引数として動的ロード関数 dlopen を呼び出すことで行う．ロード後，関数名を関数 ID から導出し，これと dlopen にて得られたハンドラを引数として dlsym 関数を呼び出す．これによって，関数実体のアドレスを取得することができるので，これを関数 ID に対応するアドレスとして，関数テーブルへ登録する．

以上の処理の後，得られた関数アドレスにジャンプし，関数実体を呼び出す．この時，呼び出し元が積んだ関数引数と戻りアドレスをそのまま関数実体に渡すため，呼び出し直前にスタックポインタを巻き戻し，マネージャが構築したスタックフレームを破棄する[⋆1]．

なお，マネージャの処理のうち，スタックポインタの操作やレジスタ経由での値の受け取りはアセンブラで記述する．

4. LLVM での実装

本章では，前章までで述べた手法を LLVM に実装する手法について述べる．

4.1 LLVM について

LLVM は，任意のプログラムを，そのライフサイクル全体を通して透過的に解析，変換するコンパイラフレームワークである[1)]．我々は，LLVM がプログラムの変換を重視している点に着目し，分割機構をプログラムの分割に LLVM のプログラム変換，最適化フレームワークである LLVM Pass Framework[2)] を用いて実装した．Pass Framework は，LLVM 独自の中間表現 LLVM IR を変換するためのフレームワークである．

4.2 前　処　理

分割機構では，プログラムの分割に LLVM Pass Framework を使用している．そこで，まずオリジナルプログラムを Pass Framework で扱うことができるようにするため，高級言語で記述されたソースコードを LLVM IR にコンパイルする（**図 7**）．この時点では，1 つのソースコードファイルは，1 つの LLVM IR 形式ファイルに対応する．

4.3 リンケージの変更

IR は，C 言語での関数と静的変数，文字列定数を併せた概念として global value を導入する．

⋆1 引数のレジスタ渡しを行う環境下では，スタック上に保存したレジスタ値をここで復帰させる．

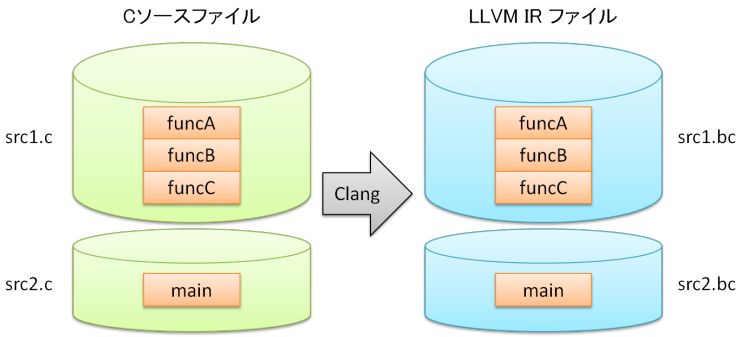

図 7 LLVM IR の生成
Fig. 7 Generating LLVM IR

している．そして，C 言語でのスコープにあたるリンケージタイプという概念があり，全ての global value は何らかのリンケージタイプを持っている．以降，本稿でも IR に倣い，関数と静的変数，文字列定数を併せて “大域値” と呼ぶ[3)]．

リンケージタイプには様々なものがあるが，プログラム分割にあたって重要になるのは，external，common，internal，private の 4 つである．external と common は C 言語での大域変数や大域関数に，internal はファイルスコープの関数や，ファイル，関数スコープの静的変数に，private は文字列定数に該当する[3)]．従って，internal と private のリンケージを持つ大域値のシンボルは外部に公開されない．

一方，3.1 節で述べたように，共有ライブラリに格納する関数シンボルは外部からアクセス可能でなければならない．そこで，分割に先立って，internal 又は private な関数を external なリンケージに変更する．これにより，起動プログラムから分割した共有ライブラリ内の関数にアクセスすることができるようになる．

4.4 変数の結合

3.2 節で取り上げたように，オリジナルプログラム中の非公開の静的変数（以降，単に非公開変数と呼ぶ）のうち，複数の関数からアクセスされるものをまとめ，一つのシンボルとオフセットで表現できる形式に変換する．

LLVM IR の Structure データ型は，メモリ上のひとまとまりになっているデータの集合を表わす型である．我々は，非公開変数をすべて格納した Structure 型公開変数を一つ生成することで，共有ライブラリから非公開変数へのアクセスを実現した．Structure 型変数の

メンバへのオフセットは，型情報から取得することができる．

この Structure 型変数の生成は，以下の手順で行う．

(1)　全ての非公開変数の型を，一つの Structure 型にまとめる
(2)　この Structure 型変数を一つ定義し，非公開変数の値を複製する
(3)　非公開変数への参照を，この Structure 型変数への参照へと置き換える

4.5　関数名の変更と分離

次に，関数に ID を付与し，その名前を ID から参照可能なものに変更する．関数名の変更した後，その関数を別の新しい LLVM IR ファイルへと分割する．この際に，ファイル名も関数名と同様に，関数 ID から導出できるものにする．また，分割された関数が元のファイル中の大域値にアクセスできるよう，分割先のファイルに元のファイルに存在する大域値の宣言を追加する．ここで宣言された大域値は，動的ロードの際に，起動プログラム中の実体へと解決される．

4.6　起動プログラムと共有ライブラリ生成

以上の手順をプログラムを構成する全ての LLVM IR に適用することにより，1 関数ごとに 1 つの LLVM IR ファイル，関数や一部の変数をを取り外した後に残った，元のファイル 1 つに対して 1 つの IR ファイル，そして共有ライブラリマネージャから構成される 1 つのオブジェクトファイルを得ることができる．最後に，これらの出力をコンパイル，リンクし，実行可能ファイルや共有ライブラリを生成する．関数ごとに生成した IR ファイルについては，単独でコンパイルして共有ライブラリに変換する．関数，変数を取り外した後に残った IR ファイルは，共有ライブラリマネージャと共にコンパイルする．これらを一つのファイルへリンクすることで，起動プログラムを得る（**図 8**）．

5.　関数のグループ化

前章まで，最もシンプルなものとして，1 関数に対して 1 共有ライブラリを生成する手法について述べた．しかし，1 関数ごとに共有ライブラリを生成すると，共有ライブラリファイルの共通部分やファイルシステム，メモリページのオーバヘッドが生じてしまい，ディスクやメモリ消費量が増加してしまう問題が生じることを確認した．そこで，我々は，関連の強い関数のグループを同一の共有ライブラリに格納することで，これらのオーバヘッドを削減することを提案する．また，前章までで取り上げた共有ライブラリマネージャは 1 関数 1 共有ライブラリを前提としたものなので，これについても，グループ化に対応する．

本章では，5.1 節で同一の共有ライブラリに格納する関数の選択方法について述べ，5.2 節で，複数の関数が格納された共有ライブラリのロードを行うための共有ライブラリマネージャの動作について示す．

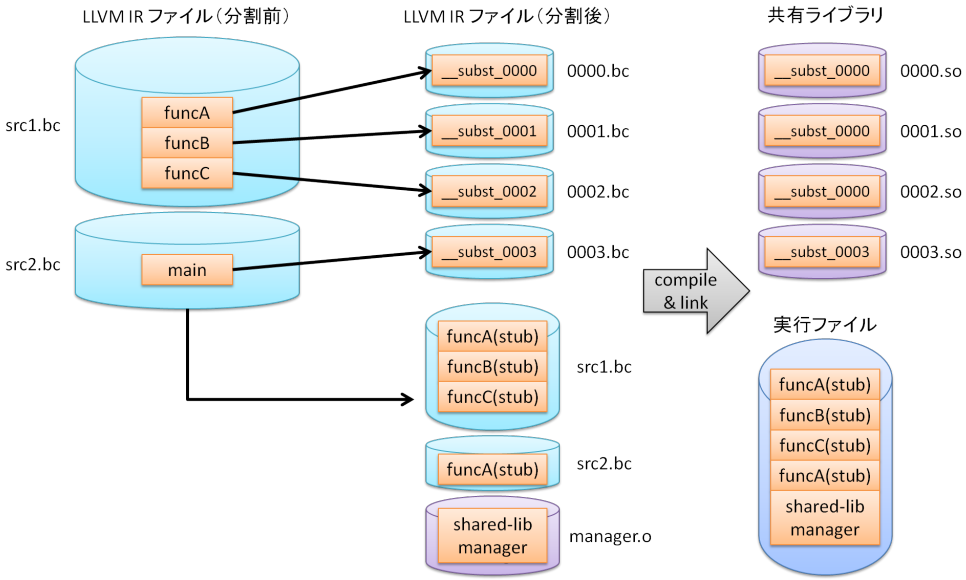

図 8　LLVM IR ファイルの分割
Fig. 8　Scattering LLVM IR

5.1　トレーシングとグルーピング

結合する共有ライブラリのグルーピングは，プログラムを実際に実行し，分割対象とする関数がはじめて実行されるタイミングを記録することで行う．この記録するための試行をトレーシングと呼び，得られた結果をトレーシング結果という．

グルーピングは次の手順で行う．

(1)　空の集合 G を生成する
(2)　トレーシング結果の先頭から，1 つ関数 F を取り出す
　(a)　G が含む関数の合計サイズと F のサイズの和がある閾値 n を超えなければ，F を G に加える
　(b)　上の条件を満たさなければ，次の処理を行う
　　(i)　現在の G を新たなグループとして定義する

(ii) 新たに空集合 G を生成する
(iii) F を G に加える
(3) トレーシング結果の末尾まで，処理（2）を繰り返す

例として，関数 a〜h が存在し，そのサイズをそれぞれ 4KiB, 8KiB, 6KiB, 48KiB, 4KiB, 4KiB, 4KiB, 4KiB であると仮定する．また，閾値を 16KiB とする．この場合，まず関数 a, b がグループ 0 に所属させる．関数 c をグループ 1 に所属させると，その合計サイズが閾値を超えるため，新たにグループ 1 を作成し，ここに関数 c を所属させる．関数 d のサイズは閾値を超えているため，単独でグループ 2 を形成する．最後に，残った関数 e, f, g, h をグループ 3 に所属させる．

グループ番号の決定後，共有ライブラリマネージャのために関数 ID からグループ ID をひくためのテーブル作成し，これをファイル（グループマップファイル）に書き出す．

関数のサイズについては，最終的な出力である共有ライブラリのものが重要となるため，前章の手順で生成した共有ライブラリを基に算出する．共有ライブラリ中の関数サイズはその共有ライブラリのサイズから共有ライブラリファイルのオーバヘッドサイズを減じることで算出する．

この方法では，閾値より大きな関数が存在しない限り，共有ライブラリファイルのサイズを一定以下に抑えることができる．さらに，トレーシング時と同じように動作させる場合，グループ順が最適な動的ロード順序となる．そこで，最初に生成したグループを 0 番として，生成順にグループに整数値を付与し，これをグループの ID 番号とした．

グループの決定後，グループごとに関数をまとめた共有ライブラリを生成する．これは，前章で生成した関数ごとの LLVM IR ファイルをリンクし，リンク後に生成された IR ファイルを共有ライブラリにコンパイルすることで行う．また，共有ライブラリファイル名は，グループ ID から導出可能なものとする．我々の実装では，“g(グループ ID の 16 進表現).so” とした．

5.2 共有ライブラリマネージャの変更

次に，結合した共有ライブラリをロードできるよう，共有ライブラリマネージャを対応させる．関数単位の共有ライブラリを使用する場合，イニシャライザは，関数 ID と関数ポインタを対応させるテーブルを用意していた．これに対し，共有ライブラリを結合した場合には，関数 ID からグループ ID をひくためのグループテーブル，グループ ID から dlopen のハンドラをひくためのハンドラテーブル，関数 ID から関数ポインタをひくための関数テーブルが必要となる．グループテーブルについては，前述のグループマップファイルをメモリ上に展開することによって得る．ハンドラテーブルは，グループと同数のハンドラを格納する領域を確保し，これを無効なアドレス値 NULL で初期化することで得る．関数テーブルについては従来と同様のものを使用する．

関数 ID を受け取ったマネージャは，まず，その ID で関数テーブルをひく．ここで有効なアドレス値が得られれば，この値を関数ポインタとして使用する．NULL であった場合，次にグループテーブルから対応するグループ ID を取得し，この ID でハンドラテーブルをひく．有効なアドレス値が得られればこのハンドラから dlsym にて関数ポインタを得，これを関数テーブルに記録する．ハンドラテーブルにも NULL が格納されていた場合，そのグループの共有ライブラリは未ロードなので，グループ ID からそのライブラリファイル名を導出し，dlopen にてロードする．この時得られたハンドラは，ハンドラテーブルに記録する．

6. 応 用 例

これまで，プログラムの分割手法と，その実装について述べた．プログラムの分割には，いくつかの応用が考えられる．本章では，応用の一つとして，ネットワーク上のプログラムの起動時間や実行時間の短縮について取り上げる．

通常，HTTP や SSH などにより，ネットワーク上からプログラムを実行する場合，そのオブジェクトファイルの全体，あるいは大部分[⋆1]をダウンロードしてから実行する．この時，ユーザは，ダウンロードが完了するまで待機する必要がある．

また，NFS のように，メモリマッピングを活用できる機構も存在する．この場合，プログラム起動時に必要な処理はオブジェクトファイルの仮想メモリ空間へのマッピングだけとなり，実際のダウンロードは，ページフォールトが発生した際に行われる．

本章では，プログラムの分割を行った場合，これらの実行方法と比較して高速に起動，実行することができることを示す．さらに，動的ロード処理をカスタマイズすることで，より高速に実行することが可能であることも示す．

6.1 基 本 動 作

プログラムを分割した場合，起動プログラムのダウンロードを完了した時点でそのプログラムの実行を開始することが可能となり，単純にネットワーク上からダウンロードした場合と比較して，起動時間が短縮することができる．また，プログラムの使用方法によって，実

⋆1 本稿で対象としている Linux では，ELF の LOAD セグメントが対象である．

行されない関数が存在するが，起動プログラムは必要となった共有ライブラリのみをロードするため，使用されなかった共有ライブラリの分，転送時間を削減することもできる．

NFS と mmap の組み合わせなどでネットワーク上のファイルをメモリにマップした場合，ページ単位でダウンロードを行うことになるが，我々の機構では，関数単位でプログラムを分割しているため，ダウンロードする範囲を細かく制御することができる．

なお，転送量そのものが大きく増加した場合はその転送時間も大幅に増加することになるので，前章の共有ライブラリファイル結合を行うことを前提としている．

6.2 プレローディングと圧縮伸長

前章の手法で共有ライブラリを結合した場合，トレーシング時と同じ処理を行わせる限り，共有ライブラリが必要となる順はそのグループ ID の順と等しい．従って，関数のトレーシング結果があれば，次に必要になるライブラリファイルを正確に予測することができる．そこで，共有ライブラリマネージャをカスタマイズし，プログラムのダウンロードをプログラム本体の実行と並列に進めることにより，起動後の実行速度の高速化する．さらに，共有ライブラリファイルを事前に圧縮することで，転送量も削減する．この場合，共有ライブラリマネージャに機能を追加し，dlopen の呼び出し前に伸長を行う．

この応用では，ダウンロードをプログラムの実行と並列に行う必要がある．また，伸長についても，マルチコア環境では並列実行可能である．そこで，実装には pthread ライブラリを用い，次のように行った．

まず，共有ライブラリマネージャを，dlopen により動的ロードを行う部分と，dlsym により関数ポインタのルックアップを行う部分（ルックアップ部）に分割した．さらに，動的ロードを行う部分を，プログラム本体と並列に動作するスレッド（動的ロードスレッド）とした．このスレッドは，グループ ID 順に共有ライブラリのロードを進め，dlopen の戻り値として得られたハンドラを，ハンドラテーブルに記録する．この時，共有ライブラリファイルが圧縮されているならば，その伸長を行う．このスレッドの起動は，初期化処理の際に行う．

ルックアップ部は，マネージャから dlopen 関数に関わる機能を外したものなので，関数テーブルとハンドラテーブルをひく処理については前章と同様である．共有ライブラリが未ロードであった場合，ルックアップ部は dlopen を呼び出す代わりに，動的ロードスレッドが該当する共有ライブラリのロードを完了するまで待機する．

表 1 評価環境
Table 1 Environment

OS	Fedora 14 32bit (Linux 2.6.35)
CPU	Intel Core2 Duo E6850 (3.00GHz)
メモリ	3.2GiB
コンパイラ	Clang 1.1
コンパイラ基盤	LLVM 2.7
最適化オプション]	（なし）
その他	strip によるシンボルの削除を実施（転送量削減のため）

表 2 評価プログラム
Table 2 Target Program

Vim-init	起動後，3 文字入力した上で強制終了
Ruby-init	空ファイルをスクリプトとして実行
Ruby-tarai	処理系に付属する竹内関数ベンチマーク
SQLite3-init	データベースファイルの生成
SQLite3-query	1000 件のレコードの追加と検索

7. 評　　価

本章では，プログラム分割の有効性を確認するため，実際にプログラムの分割を行い，生成されたファイルの評価を行う．

評価に使用した環境は，**表 1** の通りである．

また，評価には，Vim 7.3，Ruby 1.9.2-p180，SQLite3.7.6.3 の 3 つのプログラムを使用した．SQLite3 については，共有ライブラリ libsqlite3.so を評価の対象とした．

性能評価の際には，これらのプログラムに適切な入力を与える必要がある．本章では，評価用のプログラムに対し，**表 2** の処理を行うように入力を設定した．

これらのプログラムの他に，分割の効果を明確にするためのサンプルプログラムを構築し，これに対しても適用した．分割の効果を発揮しやすいサンプルとして，小さな多数の関数から構成され，実行時にはそれらのうち一部のみが呼び出されるプログラム demo1（**図 9**）を構築した．また，効果を発揮しにくいサンプルとして，実行ステップ数に対して命令数が極端に少ないプログラム demo2（**図 10**）を構築した．

7.1 関数単位分割

本節では，1 関数に対して 1 共有ライブラリを生成した際の出力ファイルと実行速度を確

```
void func0000(void) { /* some calculations */ }
void func0001(void) { /* some calculations */ }
  /* Total 65535 functions... */

int main(void)
{
  func0000();
  func0400();
  func0800();
  /* Total 64 function calls... */
}
```

図 9 サンプルプログラム 1
Fig. 9 Sample Program 1

```
int main(void)
{
  volatile unsigned i, j, k;
  for (i = 0; i < 1000000; i++)
    for (j = 0; j < 10000; j++)
      /* Null statement */;
}
```

図 10 サンプルプログラム 2
Fig. 10 Sample Program 2

表 3 関数単位分割（起動プログラムサイズ）
Table 3 Each Function to a Shared Library (Launch Program Size)

分割対象	起動プログラム [KiB]	オリジナル [KiB]	サイズ比
Vim	417	2,059	20.3%
Ruby	479	2,029	23.6%
SQLite3	125	772	16.2%
demo1	3,205	1,2291	26.1%
demo2	5	3	171.9%

表 4 関数単位分割（共有ライブラリサイズ合計）
Table 4 Each Function to a Shared Library (Total Shared Library Size)

分割対象	共有ライブラリ数	合計サイズ [KiB]	オリジナル [KiB]	サイズ比
Vim	3,984	16,193	2,059	786.6%
Ruby	4,725	15,086	2,029	743.4%
SQLite3	3,604	11,066	772	1433.4%
demo1	65,536	164,354	12,291	1337.2%
demo2	1	3	3	85.3%

認し，提案するプログラムの実用性を評価する．

プログラムを分割した際に生成されるファイルは，起動プログラムと複数の共有ライブラリファイルであるが，実プログラムを分割した際に生成されたファイルサイズを，**表 3** **表 4** に示す．この表を見ると，起動プログラムのサイズについては，オリジナルプログラムの30%未満に収まっており，分割の効果が充分に得られていることがわかる．

一方で，共有ライブラリについては，demo2 を除けばその合計サイズはオリジナルプログラムサイズの 5 倍を超えている．プログラム分割の目的にも依るが，6 章で取り上げたネットワーク越し実行など，共有ライブラリサイズの影響が大きい応用では，大きな悪影響が出ることが考えられる．

次に，実行速度については，**表 5** の通りであった．いくつかのケースでは，実行時間が相対的に大きく増加してしまっているものの，絶対的な増加時間は，高々300ms に抑えられている．このことから，分割による実行時間のオーバヘッドについては，充分に小さいと言える．

7.2 グルーピング

本節では，5 章で挙げたように，プログラムを実行した際に実行された関数のトレーシン

表 5 関数単位分割（実行速度）
Table 5 Each Function to a Shared Library (Performance)

実行プログラム	分割後 [ms]	分割前 [ms]	実行時間比
Ruby-init	84	8	1062.7%
Ruby-tarai	3,708	3,487	106.4%
Vim-init	2,289	2,022	113.2%
SQLite3-init	15	4	402.9%
SQLite3-query	34,575	34,173	101.2%
demo1	4	1	453.4%
demo2	23,927	23,488	101.9%

表 6 グルーピング分割（起動プログラムサイズ）
Table 6 A Function Group to a Shared Library (Launch Program Size)

分割対象	グルーピング [KiB]	関数単位 [KiB]	サイズ比
Vim	418	417	100.3%
Ruby	483	479	100.7%
SQLite3	125	125	100.0%
demo1	3,206	3,205	100.0%
demo2	6	5	122.7%

グを行い，この結果に基づいたグルーピングを行った．これにより，共有ライブラリごとのオーバヘッドが減り，共有ライブラリファイル合計サイズの削減を見こむことができる．

まず，生成された起動プログラムのサイズを，**表 6** に示す．起動プログラムについては，共有ライブラリマネージャの処理が増加しているものの，そのサイズは全体のサイズと比較して小さいため，関数単位の共有ライブラリ生成時と比較して，大きな差は存在しない．

次に，共有ライブラリの合計サイズを**表 7** に示す．この結果から，閾値の値を上げる，つまり共有ライブラリの総数を減らして粒度を下げることで，その合計サイズを抑えられることがわかる．

7.3 応 用 例

6 章で取り上げた，狭帯域のネットワークからプログラムを直接実行する際の性能について評価するため，プレローディングと圧縮を共有ライブラリマネージャに実装した上で，ネットワークファイルシステム上に分割したプログラムとオリジナルプログラムを配置し，性能比較を行った．グルーピングの閾値には，64KiB を採用した．狭帯域ネットワークについては，Linux の転送制御機能を用いることで疑似的に再現し，ネットワークファイルシステムには，SSHFS を採用した．

表 7 グルーピング分割（共有ライブラリサイズ合計）
Table 7 A Function Group to a Shared Library (Total Shared Library Size)

プログラム	閾値 [KiB]	共有ライブラリ数	合計サイズ [KiB]	オリジナル [KiB]	サイズ比
	32	224	3,785		183.9%
Vim	64	107	3,309	2,059	160.7%
	128	53	3,078		149.5%
	32	105	2,767		136.4%
Ruby	64	53	2,587	2,029	127.5%
	128	28	2,483		122.4%
	32	71	2,087		270.3%
SQLite3	64	35	1,967	772	254.8%
	128	17	1,895		245.5%
	32	225	8,183		66.6%
demo1	64	113	7,914	12,291	64.4%
	128	57	7,779		63.3%
	32	1	3		85.3%
demo2	64	1	3	3	85.3%
	128	1	3		85.3%

結果，実行されない関数を多数含む demo1 だけでなく，Ruby や Vim といった実用プログラムについてもほとんどのケースでオリジナルプログラムの実行時間の半分未満に抑えることができており，プログラム分割の有効性を確認することができた（**表 8**）．但し，demo2 では実行時間が増加してしまっており，1Mbps 環境下の SQLite3-query ではあまり効果が見られないことから，計算時間の長いプログラムに対しては効果が薄いこともわかる．

8. 関 連 研 究

事前にプログラムのプロファイリングを行い，実行頻度の低いコードの圧縮を行う研究[5]がなされている．この研究は，プログラムを適切な単位に分けた上でその実行頻度のプロファイリングを行い，実行頻度の低い個所を圧縮，実行時に展開することを提案している．プログラムの一定のまとまりに対し，その実行前処理を行う点で，我々の提案と関連している．しかし，この研究では，実行時に必要なメモリ量を削減するために，関数の呼び出しやリターンにおいてコード圧縮に特化した多くの処理を行っている．これに対し，我々の提案では，分割後のプログラムを実行するために必要な処理を共有ライブラリマネージャにまとめることで，実装や応用を容易にしている．また，提案手法の実装を LLVM を用いて行っ

表 8　狭帯域ネットワークからの実行
Table 8　Over a Narrow Bandwidth Network

プログラム名	帯域 [kbps]	実行時間（分割）[s]	実行時間（オリジナル）[s]	実行時間比
Vim-init	100	78.910	171.439	46.0%
	500	154.93	35.857	43.2%
	1000	79.44	18.823	42.2%
Ruby-init	100	68474	162779	41.9%
	500	13.528	32.546	41.6%
	1000	6.759	16.374	41.2%
Ruby-tarai	100	75491	166598	45.3%
	500	17.167	36.209	47.4%
	1000	10.113	19.764	51.2%
SQLite3-init	100	31.454	71.641	44.3%
	500	6.892	12.531	55.0%
	1000	3.324	71.061	44.3%
SQLite3-query	100	74.081	243.371	30.4%
	500	39.639	47315	83.8%
	1000	36402	39967	91.1%
demo1	100	273.191	681.547	40.1%
	500	54.701	136.171	40.2%
	1000	27.331	68.059	40.2%
demo2	100	25.181	23.756	106.0%
	500	24.163	23.557	102.5%
	1000	24.048	23.547	102.1%

ているため，他のプラットフォームへの移植も考えることができる．

仮想メモリ機構において，必要となるメモリページを予測し，事前に主記憶装置やキャッシュメモリに格納する研究が数多く行われている．予測には，マルコフモデル[4] などが採用されている．これらの研究は，プログラムの一部を，事前に低速な記憶装置から高速なものに読み込む点で，我々の提案と類似している．しかし，これら研究を実装する場合，OS やハードウェアに一方で，我々の分割機構ではプログラムを関数単位で管理するので，ロードする部位を，仮想記憶におけるページサイズよりも細かく制御することが可能である．これにより，応用例で挙げたように，ネットワーク上からプログラムを実行する際に高い効果が期待できる．

プログラムの実行状況に応じて，キャッシュメモリの階層構造を動的に変化させる研究[6]，ハードウェアの再設定を行う研究[7]，主記憶装置上のデータを圧縮する研究[8] がある．これらの研究は，いずれもプログラムの実行フェーズに基づいて，ハードウェアやデータの操作を行うものである．本稿では，プログラム分割の応用例として，ネットワークからプログラムを実行するケースを取り上げた．我々の提案した応用例では，プロファイリング時と同じプログラムを実行することを前提としてきたが，この論文の手法を応用することで，幅広い実行方法に対応させることが可能となると考えられる．

9. おわりに

本稿では，プログラムを起動プログラムと複数の共有ライブラリに分割する手法と，その実装について述べた．我々の手法で，1 つの関数に対して 1 つの共有ライブラリを生成することや，関連の強い分割した関数同士を結合した上で共有ライブラリを生成することが可能である．

実際にプログラムを分割したところ，1 つの関数に対して 1 つの共有ライブラリを生成した場合，共有ライブラリサイズの合計がオリジナルプログラムと比較して大きく増加する．この点が問題となる場合，実際の関数の実行順をトレーシングした上で関数のグルーピングを行い，そのグループごとに共有ライブラリを生成することで対応することができることも示した．

また，応用として，狭帯域ネットワーク上から分割されたプログラムを実行する例を取り上げた．プログラムが分割されていれば，実際に必要となった部分のみダウンロードするだけで，プログラムを実行することが可能であることを示した．さらに，プレダウンロードや共有ライブラリ圧縮で，さらなる高速化も見込めることを示した．実際に，ネットワークファイルシステム SSHFS から単純に実行した場合と比較し，良いものでは 60%の実行時間削減を実現した．また，NFS 上からプログラムを実行した場合実行ファイルは仮想メモリ空間へマッピングされるが，これに対しても遜色ない性能を実現した．

次に，今後の課題について述べる．まず，トレース時と異なる実行順序での性能を確保を挙げる．我々は，関数のグルーピングを，実行時のトレーシングに基づいて行った．この方法では，プログラムがトレーシング時と異なる方法で使用された場合に対応することができず，不要な関数がロードされてしまい，実行時性能低下の原因となることが考えられる．

また，応用例として，共有ライブラリのプレローディングについて取り上げた．プレローディングすべき関数はトレーシング結果に基づいて予想しているが，現時点での我々の実装では，外れた際に対応することができず，パフォーマンス低下の大きな原因となる．そこで，予想が外れた際に，ペナルティを最小限に抑えることが課題として挙げられる．

参 考 文 献

1) Chris Lattner, Vikram Adve: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization,p.75, March 20-24, 2004, Palo Alto, California
2) Chris Lattner, Jim Laskey: *Writing an LLVM Pass*, http://llvm.org/docs/WritingAnLLVMPass.html
3) Chris Lattner, Vikram Adve: *LLVM Language Reference Manual*, http://llvm.org/docs/LangRef.html
4) Doug Joseph, Dirk Grunwald: *Prefetching using Markov Predictors*, ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture
5) Saumya Debray, William Evans: *Profile-Guided Code Compression*, PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation
6) Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, Sandhya Dwarkadas: *Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures*, Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (2000)
7) Ashutosh S. Dhodapkar, James E. Smith: *Managing Multi-Configuration Hardware via Dynamic Working Set Analysis*, ISCA '02 Proceedings of the 29th annual international symposium on Computer architecture
8) Doron Nakar, Shlomo Weiss: *Selective Main Memory Compression by Identifying Program Phase Changes*, WMPI '04 Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture