Vol.2012-SLDM-155 No.21
Vol.2012-EMB-24 No.21
2012/3/3

オブジェクトファイルの細分化機構と応用

河合夏輝[†1]　笹田耕一[†1]

我々は，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ことで，プログラムのロード処理をより柔軟に制御，効率化する手法を提案してきた．本稿では，この機構によって分割されたプログラムのパフォーマンス向上やメモリ消費量削減を実現するための手法を取り上げる．我々が提案する分割機構は，関数を最小単位としてプログラムを分割することでより柔軟に制御することを可能とし，プログラムのロード処理の効率化を実現するものであった．一方で，プログラムの実行を通して分割のオーバーヘッドが生じ，パフォーマンス低下が生じる問題が残存していた．これに対し，本稿では，このオーバーヘッドを削減する手法を提案する．また，不要になった共有ライブラリをメモリ上からアンロードすることにより，メモリ消費量削減をさらに進める手法についても実現した．本稿では，分割機構の概要を示した上で，これらの手法について述べ，その評価と今後の課題について議論する．

Object Files Scattering System and its Application

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

We have proposed a system scatters a program to a launch program and many shared libraries to control the loading process of the scattered program more flexibly and effectively. This paper describes methods to improve the performance or the memory consumption of programs scattered by the system. The system increases flexibility of programs by scattering them in functions unit and makes the loading process of the program effectively. On the other hand, the scattered programs has performance overhead of scattering. In this paper, we propose a method to reduce the overhead and increase the performance of the scattered programs. Moreover, we propose a method to reduce memory consumption by unloading unnecessary shared libraries from memories.

In this paper, we describe the summary of the scattering system and the methods. Then, we show the evaluation of the method and discuss problems to be solved.

1. はじめに

ソフトウェアを開発する際には，プログラムを適切な単位のモジュールに分割して様々な形のモジュールを作成し，それらをリンカによって 1 つに結合する．モジュールの例として，多くの個所で実行される処理をまとめたサブルーチンや関数といったものが挙げられる．プログラムを複数の共有ライブラリに分割し実行時に動的リンクすることもできるが，分割単位は適切に指定する必要がある．そのため，共有ライブラリの設計者は，分割対象のプログラムの内部構造や機能を十分に把握していなければならず，手間がかかる．

そこで，我々は，プログラムを自動でモジュールに分割することができる分割機構を提案している[1)]．分割機構を使用した場合，プログラムの構成や機能を事前に把握する必要がないため，手動でモジュール構成を決めて分割する場合と比較して，容易かつ高速にモジュール分割することができる．一方で，この分割機構はパフォーマンスやメモリ使用効率の課題を抱えていた．本稿では，提案している分割機構の概要と抱えていた課題を示し，この課題に対する改善手法を提案する．さらに，この手法に対する評価を行い，今後の課題について議論する．

分割機構は，C のソースファイルを入力として受け付け，1 つの起動プログラムと複数の共有ライブラリを出力する．この起動プログラムは，実行時に必要となった共有ライブラリをロード，実行することで，分割前のオリジナルプログラムと全く同じ機能を実現する．分割の際，分割機構内部では，プログラムを一度関数単位で分割した後に，これらを適切に組み合わせることで共有ライブラリを生成している．分割の単位に，関数という細粒度のものを用いることで，モジュールの構成を柔軟に決めることが可能である．なお，分割機構の実装は，多くの環境に対応し，最適化のためのフレームワークが充実した LLVM[2)] を用いて行っている．

分割機構が生成する起動プログラムには分割前のプログラムに含まれる関数の代わりとなる小さなスタブが含まれている．ユーザから見ると，このスタブははオリジナルプログラム中の関数と完全に同じ振る舞いをするように見えるため，分割後のプログラムのユーザは分割されていることを意識せずに使用することができる．さらに，未使用の関数が呼び出されて共有ライブラリがロードされるタイミングで，任意の処理を追加することも可能であ

†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

Vol.2012-SLDM-155 No.21
Vol.2012-EMB-24 No.21
2012/3/3

る．これにより，共有ライブラリを事前に圧縮しておき，必要に応じて伸長する等の機能を実現することができる．

分割機構の応用例としては，ストレージやメモリ資源の限られた環境で使用するケースが挙げられる．仮想メモリを持たない環境では不要になったコードをスワップアウトさせることができない．このような環境下でも，プログラムを分割し，ロード，アンロードを機能ごとに行うことができれば，ユーザがある時点で必要としている機能以外をメモリ上から追い出すことが可能となる．また，ストレージ容量の制約が強い場合，前述のようにプログラムの圧縮伸長機能の追加も有効に働く．

2. 分割機構の全体像

本章では，我々の先行研究における分割機構[1] について，その全体像を示す． まず，2.1 節にて分割機構そのものについて示し，2.2 節では分割後のプログラムの動作について述べる．さらに，2.3 節にて，先行研究にて残存していた課題について触れる．

2.1 分割機構の動作

通常，高級言語で記述されたプログラムをビルドして実行ファイルを得る場合，まずソースファイルをオブジェクトファイルに変換して，このオブジェクトファイルをリンクして 1 つの実行ファイルを生成する．

これに対し，分割機構を使用する場合，まず全てのソースファイルを分割機構のコマンドラインインターフェース（CLI）からコンパイル，分割して，1 ソースファイルにつき 1 つ生成される起動プログラム用中間ファイルと，1 関数につき 1 つ生成される共有ライブラリ用中間ファイルを生成する．全てのファイルの分割が終わった後，起動プログラム用の中間ファイルを機械語オブジェクトファイルにコンパイル，リンクすることで起動プログラムを生成する．最後に，共有ライブラリ用の中間ファイルをコンパイル，リンクし，複数の共有ライブラリを生成する．

また，分割に先立ってプログラムを実行し，関数の呼び出し順序を記録する．そして，この記録に基づいて関数のグルーピングを行い，最初に実行されるタイミングの近い関数をまとめて 1 つの共有ライブラリに格納する．

2.2 分割後のプログラムの動作

分割されたプログラムの利用者は，元のプログラムの代わりに，生成した起動プログラムを実行する．起動プログラムは，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．起動プログラムには，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が 1 つと，元のプログラムに存在する関数に 1 対 1 で対応する，共有ライブラリマネージャを呼び出す小さな関数（スタブ）を持つ．

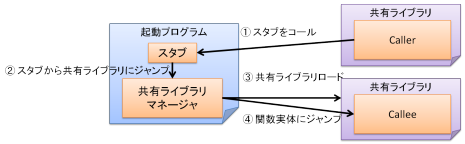

図 1 分割後のプログラムの動作
Fig. 1 Behavior of a Scattered Program

スタブは，分割対象となった関数の代わりとなるもので，呼ばれた関数の情報を共有ライブラリマネージャに渡すだけの役割を持つ．そして，共有ライブラリマネージャは関数実体に処理を渡すための関数で，1 つの起動プログラム中に 1 つ存在する．共有ライブラリマネージャは，適切に共有ライブラリを管理し，分割された関数本体を実行する（**図 1**）．

このように，スタブと共有ライブラリマネージャは最終的にオリジナルプログラム中の関数に処理を渡すので，関数を呼び出す側から見ると，実行時間等の性能を除けばオリジナルプログラムの関数との違いはない．

さらに，共有ライブラリマネージャには関数が呼び出されるタイミングで実行する任意の機能を追加することもできる．この機能は，共有ライブラリファイルの圧縮等の応用に活かすことができる．

2.3 先行研究の課題

先行研究における分割機構では，次の 2 点の問題が解決していなかった．

- 一度ロードした共有ライブラリをアンロードすることができない
- 関数呼び出しの度にスタブ呼び出しのオーバーヘッドが生じる

分割機構を仮想メモリを持たない環境に応用する場合，一度ロードした共有ライブラリをアンロードすることでメモリ消費量の削減を期待することができる．一方で，先行研究の分割機構では，共有ライブラリのアンロードを実現することができていなかった．そこで，本稿では，一度ロードした共有ライブラリについてアンロードの可否を導出するアルゴリズムによって，共有ライブラリのアンロードを実現する手法について述べる．なお，アンロードを行う場合，1 つの共有ライブラリに含める関数の組み合わせの決定（関数グルーピング）に従来とは異なる戦略を用いる必要がある．

また，先行研究の分割機構にて分割したプログラムは，スタブと共有ライブラリマネー

Vol.2012-SLDM-155 No.21
Vol.2012-EMB-24 No.21
2012/3/3

ジャを関数呼び出しの度に実行することでオーバーヘッドを生じさせていた．このオーバーヘッドの削減も，先行研究の課題であった．

3. 共有ライブラリのアンロード

3.1 アンロード手法

先行研究[1)]では，一度ロードした共有ライブラリのアンロードを考慮しなかった．仮想メモリを使用することのできる環境では不要となったメモリページが自動でスワップアウトされるためアンロードの必要はないが，仮想メモリのない環境では不要となった共有ライブラリのアンロードを行うことでメモリ消費量を削減することができる．

アンロードを行う際に注意する点として，呼び出し元の関数がアンロードされてしまうと呼び出し先からリターンすることができなくなるため，コールスタック上の関数を含む共有ライブラリはアンロードできないことが挙げられる．また，C 標準ライブラリには setjmp, longjmp 関数というスタック操作を行う関数が含まれているため，これらの関数によってコールスタックが巻き戻った場合に対応する必要がある．

そこで，我々は，マシンスタックの他に独自のスタック（共有ライブラリスタック）を共有ライブラリマネージャ上に用意し，ある関数が必要になったタイミングでのマシンスタックポインタの値を記録することで不要となった共有ライブラリを検出する手法を提案する．スタックには，関数 ID と，その関数が必要になったタイミングを示すスタックポインタ値の 2 つを格納する．

さらに，同じ共有ライブラリに格納された関数のうち使用中のものの数を格納するためのテーブルも用意する．このテーブルのサイズは共有ライブラリ数と等しく，各エントリの初期値は 0 とする．マルチスレッド環境では，テーブルの操作を行う際には排他制御を行う．

共有ライブラリスタックへのプッシュは，共有ライブラリマネージャが関数アドレステーブルに記録するタイミングで行う（**図 2**-1）．共有ライブラリマネージャは関数 ID とその時のスタックポインタ値の組を，関数アドレステーブルに記録するのと同時に共有ライブラリスタックにプッシュする（図 2-2）．さらに，該当するリファレンスカウンタに 1 加える（図 2-3）．

この処理とは別に，共有ライブラリマネージャが呼び出された際に，共有ライブラリスタック最上位のスタックポインタが現在のスタックポインタよりも上にあるか同じであれば（**図 3**-1），ユーザスタックからのポップを行う（図 3-2）．そして，その関数は既に使用されていないので対応する共有ライブラリのリファレンスカウンタの値を 1 減らす（図 3-3）．

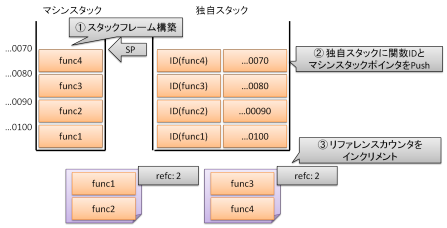

図 2 関数呼び出し時の動作
Fig. 2 Calling Functions

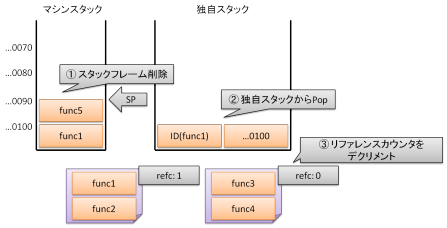

図 3 関数リターン時の動作
Fig. 3 Returning from Functions

リファレンスカウンタの値が 0 になった場合は共有ライブラリが不要になったと判断できるため，アンロードを行う．

なお，この手法はマシンスタックのアドレスを活用しているため，マシンスタックを独自に操作するプログラムに対しては使用することができない．

3.2 関数グルーピング

我々の先行研究では，共有ライブラリファイルのヘッダ等のオーバーヘッドを削減しつつ，ロード処理を効率的に行うために，複数の関数を 1 つの共有ライブラリに格納していた．そして，1 つの共有ライブラリに含める関数群は，最初に実行されるタイミングが近い関数をグループにすることで決定していた．

しかし，共有ライブラリのアンロードを行う場合，先行研究とは異なる手法でグループを決める必要が生じる．前節の手法では，アンロードを使用中の関数が 1 つでもあればその共有ライブラリをアンロードすることができない．そこで，共有ライブラリのアンロードを行う場合，関数が最後にリターンするタイミングが近い関数を 1 つの共有ライブラリに格納

Vol.2012-SLDM-155 No.21
Vol.2012-EMB-24 No.21
2012/3/3

することで，アンロード処理を効率化する．

4. 性能オーバーヘッドの削減

先行研究の手法にて分割したプログラムは，全ての共有ライブラリがロード済みであったとしても，毎回スタブと共有ライブラリマネージャが呼び出されていた[1)]．実行速度を求められる場面では，これらの処理による性能のオーバーヘッドが問題となることが考えられる．

そこで，本節では，前節までで述べたテーブルとは別の関数アドレステーブル（高速直接参照テーブル）を用意し，これを使うことで，関数ポインタを経由しない関数呼び出しの速度を高速化する手法について述べる．

高速直接参照テーブルの初期値は，ID に対応するスタブのアドレスとする．そして，分割の際に，最終的にリンクされる全てのプログラム中でプログラムの中で分割対象となる関数を呼び出している箇所を，高速直接参照テーブルの対応するエントリによる間接呼び出しに変更する．これにより，全ての関数呼び出しは，高速直接参照テーブルを経由することになる．また，共有ライブラリマネージャの処理も変更する．共有ライブラリマネージャが共有ライブラリをロードし，関数アドレスを取得した際に，高速直接参照テーブルの対応するエントリも取得した関数アドレスに変更する．これにより，その関数は次回からスタブや共有ライブラリマネージャを経由せずに呼ばれるようになる．

ただし，この手法では，ポインタ変数を使用した間接呼び出しを使用している個所に対しては効果が得られない．そこで，間接呼び出しを多用したプログラムに向けて，スタブから共有ライブラリマネージャを呼び出す際にも同様のテーブルを導入し，共有ライブラリマネージャのオーバーヘッドを削減する．この間接呼び出し対応の有無によるパフォーマンスの変化についても，5 章にて評価する．

なお，本節で取り上げた手法を共有ライブラリのアンロードと同時に用いると，同じ関数を繰り返し呼び出すようなケースにおいて，共有ライブラリのアンロードが正常に動作しない可能性がある．そのため，この 2 つの手法を併用することはできない．

5. 評　　価

5.1 関数テーブルによる高速化

本節では，関数テーブルによる性能面でのオーバーヘッドの削減効果についての評価を行う．評価環境は**表 1** の通りである．また，分割対象には言語処理系 Ruby 1.9.2 とテキストエディタ Vim 7.3 を採用し，**表 2** のベンチマークを実行することで評価した．

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	（指定なし）
その他	strip によるシンボルの削除を実施

表 2 評価プログラム
Table 2 Benchmark Programs

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

表 3 関数テーブルによる高速化 [ms]
Table 3 Acceleration with Function Table

実行プログラム	間接参照対応	間接参照未対応	関数テーブルなし	オリジナル
Ruby-init	83.5	84.4	84.3	7.2
Ruby-tarai	2,832.2	3,312.9	3,345.4	2,810.9
Vim-init	2,276.3	3,422.5	3,426.4	2,019.6

評価結果として，各ベンチマークプログラムを実行してから終了するまでの時間を測定した．1 回の実行時間が 1 秒を下回るものについては 1000 回，上回るものについては 10 回実行し，その平均を算出した．

結果を見ると，ポインタによる間接参照に対応した場合は，Ruby-init 以外で実行速度が改善していることがわかる．Ruby-init で効果が現れていないのは，共有ライブラリのロード時間の影響が大きいためだと考えられる．

5.2 共有ライブラリアンロード

本節では，共有ライブラリのアンロードによるメモリ消費量の削減効果についての評価を行う．共有ライブラリのアンロードの実装に用いた環境は，仮想メモリを使用するので，メモリ消費量削減効果を確認することが難しい．そこで，評価環境において関数のロード，アンロードのタイミングと共有ライブラリのサイズの情報を取得し，これに基づいたシミュレーションを行った．また，本節では評価対象として，小型でハードウェアへの制約が強い

Vol.2012-SLDM-155 No.21
Vol.2012-EMB-24 No.21
2012/3/3

環境でも活用しやすい言語処理系 Lua 5.1.4 を採用し，付属テスト全てと空のテストスクリプト empty を実行した．ただし，テスト life についてはスクリプトに変更を加え，繰り返し回数を 20 回に減らした．入力データを求めるものについてはそのスクリプトファイル自身を入力とした．また，プログラム間の比較のため，前節で使用していた Vim-init も対象に加えた．

評価対象は，ロードされている共有ライブラリサイズ合計がプログラムの実行から終了までの間で最大となった時の値とした．最大値を評価の用いるのは，メモリへの制約が強い環境ではメモリの最大消費量が問われることが多いためである．また，比較対象として，生成される共有ライブラリ合計の値も算出した．共有ライブラリのサイズは 32KiB 程度となるようにした．

結果，表 4 の通りの値を得た．Lua に関しては最もメモリ消費が少ない空スクリプト実行では共有ライブラリによるメモリ消費を 25%程度まで抑えられている一方，60%近くの共有ライブラリをロードしているケースもある．Vim-init では 5.3%と，Lua と比較して良好な結果が得られている．このように，アンロードによるメモリ消費削減効果は分割対象のプログラムや入力によって大きく異なるため，応用する場合はシミュレーション等での見積もりが必要になると考えられる．

以上のように，アンロード処理によるメモリ消費削減効果はプログラムや入力によっては大きな効果を挙げるが，そのばらつきも大きいことが判明した．従って，分割機構をメモリ消費量削減を狙って応用する場合，事前にシミュレーション等を行いその効果を確認することが望ましいと言える．

6. 議　　論

本稿では，我々が提案している分割機構のいくつかの課題について，解決方法とその評価について示した．本章では，この分割機構のさらなる改善手法について議論する．

本研究では，実装の容易性および移植性を考慮して，動的リンカ，ローダには既存のライブラリを用いた．しかし，既存のライブラリは様々な用途に使用できるように多くの機能が含まれているため，性能や共有ライブラリファイルサイズにおけるオーバーヘッドが大きいという問題がある．これに対し，メモリやストレージ容量に対する要求が厳しい環境では，分割機構に特化した独自の動的リンカ，ローダを用意することで，共有ライブラリのサイズやロードされるデータ量を削減することを考えることができる．本節では，分割機構専用の動的リンカ，ローダを導入することによる効果について論じる．

表 4　共有ライブラリによるメモリ消費
Table 4　Memory Consumption of Shared Libraries

ベンチマーク名	最大ロードサイズ [KiB]	共有ライブラリ合計 [KiB]	削減率
Lua bisect	159	356	44.6%
Lua cf	158	356	44.5%
Lua echo	127	356	35.7%
Lua env	158	356	44.3%
Lua factorial	158	356	44.3%
Lua fib	158	356	44.3%
Lua fibfor	179	356	50.3%
Lua hello	127	356	35.7%
Lua life	182	356	51.1%
Lua luac	126	356	35.3%
Lua printf	159	356	44.6%
Lua sieve	211	356	59.4%
Lua sort	159	356	44.8%
Lua trace-calls	158	356	44.4%
Lua trace-global	157	356	44.1%
Lua xd	158	356	44.2%
Lua empty	95	356	26.8%
Vim-init	336	6,385	5.3%

ここまで見てきた設計，実装を踏まえると，分割機構においては，動的リンカ，ローダには次に挙げる 3 つの役割があると考えられる．

- 共有ライブラリに含まれる関数やデータをメモリにロードする
- 関数が使用するデータのアドレスを解決する
- 各関数の先頭アドレスを取得する

まず，共有ライブラリの中に含まれている命令列やデータを，メモリにロードする機能が挙げられる．実行するプログラムがメモリ空間上に配置されていなければ実行することはできないため，ロード機能が必要となるのは自明であり，分割機構に特化した動的ローダでも効率化することはできない．

次に，各関数が使用するデータのうち，起動プログラムに含まれているもののアドレスを解決する必要がある．この機能は，複数の関数からアクセスされるデータを起動プログラム中に配置するものの，起動プログラムが配置されるアドレスを事前に知ることができないという制約上，不可欠なものである．1 つの共有ライブラリを，多くのプログラムがロードして使用する通常のケースでは，そのライブラリを使用するプログラムが事前にはわからないため，動的ロードの際に全てのデータのシンボルをテーブルから引き，解決する必要があ

る．一方，分割機構が生成する共有ライブラリは，起動プログラムのみからロードされることが事前に判明している．そのため，起動プログラムにおいて全ての変数を 1 つの連続した領域に格納すると，共有ライブラリにはその領域の先頭アドレスさえ知っていれば全てのデータにアクセスすることができるようになる．この場合，解決すべきアドレスは 1 つだけなので，分割機構に特化した動的ローダを作成することでロード処理を高速に行うことができる．さらに，起動プログラムのシンボルテーブルのうちデータを指すものについては削除することができるため，ストレージやメモリ消費も改善することができると考えられる．

さらに，各関数の先頭アドレスを取得する際に，汎用の動的リンカ，ローダでは文字列でのみ指定可能であることが多い．これに対し，分割機構では全ての関数に必ず ID が付与されているため，文字列よりも短い数値で各関数にアクセスすることができる．これにより，ファイルサイズを若干削減することができる．

7. 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[3]がなされている．この研究では，プログラム中の実行頻度の低い関数を圧縮し，実行時に伸長することを提案している．また，伸長した関数をメモリから削除する手法も提案している．プログラムを関数単位で取り扱い，圧縮等の処理を行う点が我々の研究と類似しているが，我々の提案する分割機構は設計をシンプルにすることで，先行研究や本稿で取り上げたような応用を容易に実現することができる．また，実装を LLVM を用いて行っているために他のプラットフォームへの移植も容易である．一方で，この研究ではコード圧縮に特化した様々な手法を用いている．これらの手法の分割機構への応用や評価は，今後の課題である．

グループ構成の決定に関連した研究として，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[4]，ハードウェアの再設定を行う研究[5]，主記憶装置上のデータを圧縮する研究[6]，コールスタックの内容に基づいて攻撃を検知する研究[7]が挙げられる．これらの研究では，いずれもプログラムの実行フェーズに基づいて，様々な操作を行っている．先行研究や 3.2 節で取り上げたグルーピング手法では，事前に関数実行順序のトレーシングを行った時と大きく異なる入力に対応できないが，この論文の手法を応用することで幅広い入力に対応させることが可能となると考えられる．

8. 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機構について，そのパフォーマンスやメモリ消費削減効果を向上させる手法について示した．評価では，実際に取り上げた手法を用いて分割したプログラムのパフォーマンス測定を行い，実際にプログラムの実行時間を短縮することができていることを確認した．

また，メモリ消費量についてはシミュレーションによって効果の確認を行った．結果，良い場合は共有ライブラリによるメモリ消費を 9 割以上削減することができることが判明した．但し，プログラムやベンチマークによってばらつきは大きいので，実際に活用する場合は事前に効果を見積もることが必須である．

6 章では，動的ローダについて取り上げた．我々の分割機構によって分割したプログラムは，汎用の動的ローダを用いて動作する．これは，移植性向上や実装コスト低減のためであるが，分割専用の動的ローダを導入することで動的ロードのオーバーヘッド削減を期待することができる．この専用ローダの実装と評価が今後の課題である．

参 考 文 献

1) 河合 夏輝, 笹田 耕一: *LLVM* **を用いたオブジェクトファイルの細分化**, 情報処理学会研究報告, Vol.2011-OS-118 No.22, 2010
2) 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
3) Saumya Debray, William Evans: *Profile-Guided Code Compression*, PLDI '02 Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation
4) 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)
5) 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
6) 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
7) 神山 貴幸, 大山 恵弘: **コールスタック情報を利用したモデル分割に基づく異常検知システム**, 情報処理学会論文誌. コンピューティングシステム 2(1), 12-22, 2009-03-25