Javaの自動並列化における例外フローとメソッドディスパッチのインライン化解析

田端啓一[†1]　木村啓二[†1]　笠原博徳[†1]

本稿では，Java プログラムを自動並列化するためのコンパイル手法を提案する．Java プログラムから複数粒度の並列性を抽出する上では，2 つの動的なメカニズムに対する解析の複雑さが問題となる．1 つは，例外によって生じる制御フローの解析である．もう 1 つは，メソッド呼び出しによって生じる動的ディスパッチの解析である．本稿の提案手法は，ランタイム環境におけるこれらの動的なメカニズムを，中間表現でのプリミティブな条件分岐にインライン展開し，解析を容易にする．提案手法を実装し評価したところ，Java で記述された optical flow など 3 つの画像処理プログラムから並列性の抽出が可能となり，IBM Power5+ 8 プロセッサにおける 1 プロセッサに対する速度向上率として，最低 7.84 倍の性能向上が得られた．

Inlining Analysis of Exception Flow and Method Dispatch on Automatic Parallelization of Java

Keiichi Tabata,[†1] Keiji Kimura[†1]
and Hironori Kasahara [†1]

This paper proposes compilation methods for automatic parallelization of Java. Java programs have two dynamic mechanisms which complicates multiple-grain parallelism extraction. The one is implicit or possible control flow by exception. Another one is dynamic dispatch for virtual method call. The proposed methods inline these dynamic mechanisms into primitive conditional branches on intermediate representation for easier analysis. The evaluation result shows at least 7.84x speedup on optical flow and other two image processing programs with IBM Power5+ 8 processors.

†1 早稲田大学 基幹理工学研究科 情報理工学専攻
Graduate School of Computer Science, Waseda University

1. はじめに

携帯電話をはじめとする組み込みプラットフォームにおいて，Java アプリケーションが移植性の高さを理由に利用されている．ところが，このようなアプリケーションで高速なメディア処理を必要とする場合には，移植性を犠牲にして，Java 仮想マシンと連携するネイティブコードの呼び出しを行う必要がある．その理由は処理性能である．組み込み向けプロセッサは，消費電力や発熱の問題から，動作周波数が低く抑えられている．このため，高速なメディア処理を行うには，SIMD 命令などを用いて 1 クロックのレベルでチューニングを行うことが，現状では必要である．JIT コンパイルによってそのような最適化を達成することは困難である．よって，予めチューニングされたネイティブコードが必要となる．ネイティブコードは移植性を妨げるものであるため，できる限り用いないことが望まれる．

ところで，近年，組み込みプラットフォームにおいて，マルチコアプロセッサの採用が広がっている．マルチコアプロセッサを活用することにより，組み込みプラットフォームにおいて，低い動作周波数で高い処理性能を得ることが可能である．このことから，マルチコアプロセッサでは，高速なメディア処理を，ネイティブコードを用いずに，Java で記述可能となることが期待できる．

一般に，マルチコアプロセッサのような並列アーキテクチャ上で，性能のよいプログラムを効率よく開発することは容易でない．このため，自動並列化コンパイラによる支援に期待が高まっている．Java においては，ループ並列化がすでに実現している[1)]．しかしながら，増え続けるプロセッサコア数を活用するためには，プログラム全域から複数粒度の並列性を階層的に抽出することが必要である．これは Java において実現していない．

筆者らの先行研究である OSCAR 自動並列化コンパイラ[2)3)] では，FORTRAN77 と C において，複数粒度の並列性を階層的に抽出する自動並列化が実現している．これを Java に適用したところ，2 つの問題が生じた．1 つは，例外フローによる制御依存が並列化を阻害することである．もう 1 つは，メソッドの多態性によってコールグラフの解析が煩雑となることである．

本稿では，まず，2 章において不要な例外フローを除去して並列化を可能とする手法を提案する．次に，3 章においてコールグラフ生成を簡略化する手法を提案する．そして，4 章においてそれぞれの手法の評価を示し，最後に 5 章でまとめを述べる．

```
array = new int[100];
for(int i=0; i<100; i++)
  array[i] = 100 / (i-75);
```

図 1　ループ内の例外 (暗黙的な例外の場合)
Fig. 1　exception in loop (implicit case)

```
try{
  readA(); // may throw exception
  readB(); // may not be reached
} catch(...)  {
```

図 2　メソッド呼び出し間の例外 (潜在的な例外の場合)
Fig. 2　inter-method exception (possible case)

2. 例外フローの除去

本章では，実行時の例外処理メカニズムを，中間表現でのプリミティブな条件分岐としてインライン展開し，これに定数伝搬とデッドコード削除を適用することで，例外フローを除去する手法について述べる．

2.1 例外が並列化に与える影響

プログラム中で，例外がスローされる可能性のある区間は，投機的な手法を用いない限り，並列化することができない．つまり，あるプログラム区間を並列化するためには，その区間に例外のスローが存在しないことを，解析によって見出さなければならない．

例外のスローによって並列化が適用できなくなる例として，図 1 と図 2 を挙げる．図 1 では，明らかに，i=75 のとき 0 による除算が発生する．つまり，i=75 のときループ外へのジャンプを有するので，このループは do-all 並列化できない．また，図 2 では，readA() の呼び出しの内部で例外がスローされる可能性があり，readB() のステートメントに到達しない可能性がある．このため，readA() の内部を解析して例外のスローが存在しないことを確認しなければ，readA() と readB() に対してメソッド間の粗粒度並列化を適用することができない．

2.2 例外解析の先行研究

Artigas らによる Java のループ並列化の研究[1] では，オブジェクトの null チェックおよび配列境界チェックを行う条件分岐をループの外側に追加し，並列化できる場合と並列化できない場合でマルチバージョン化を行う手法が提案されている．

また，Java に限らず，配列境界チェック[4] や，配列境界例外の除去と最適化[5] については多くの研究がなされている．特に，添字がループのインダクション変数である場合について，配列境界チェックの一部を除去する手法が知られている[6]．また，Sun Microsystems による Java 仮想マシン実装においても，配列境界チェックに対して，添字の最小値，最大値を用いた最適化が行われている[7]．C++においては，インタープロシージャの例外制御フローの解析と最適化が研究されている[8]．

```
// 元のステートメント
array[subscript] = x;
// 変換後のステートメント
if (array == null)
 goto handler;
if (subscript < 0 ||
 subscript >= array.length)
 goto handler;
array[subscript] = x;
```

図 3　配列アクセスごとの例外チェック
Fig. 3　exception check for array access

```
// 元のステートメント
foo();
// 変換後のステートメント
ExcInfo exc;
exc.occured = false; // DEF
foo(&exc); // may KILL
if (exc.occured) // USE
 goto dispatcher;
```

図 4　メソッド呼び出し間の例外チェック
Fig. 4　exception check for inter-method catch

2.3 例外フローを除去する提案手法

並列化可能なプログラム区間を見出すために，不要な例外フローを除去する必要がある．しかし，その解析精度とコンパイル時間はトレードオフの関係となる．よって，並列性を抽出するために必要となる精度を保ちつつも，できるだけ簡便な手法が求められる．

提案手法は，実行時の例外処理メカニズムを，中間表現でのプリミティブな条件分岐にインライン展開するものである．これにより，定数伝搬とデッドコード削除といった単純な最適化によって，不要な例外チェックによる制御フローを除去し，並列化を可能とする．

まず，配列アクセスの例外チェックについて述べる．図 3 のように配列アクセスを変換し，実行時の例外チェックを if 文として埋め込む．このとき，配列の長さが定数伝搬され，かつ，添字 subscript が解析可能な形式であれば，if 文の条件式をコンパイル時に畳み込み可能である．結果として，不要な例外チェックは，単純なデッドコード削除によって除去される．あるループにおいて例外チェックがすべて除去されれば，そのループを並列化するにあたって，例外による障害はなくなる．なお，添字多項式の解析を，どれだけの精度で行うべきかという問題がある．筆者らの実装では，添字多項式がループインダクション変数のアフィン関数である場合に，最小値と最大値で比較するという簡便な解析を行った．これには定数畳み込みに付随して高速に実行されるという利点がある．

次に，メソッド呼び出し間の例外チェックについて述べる．スローされた例外がメソッド

表 1　メソッド呼び出しを行う Java バイトコード命令
Table 1　Java bytecode ops for method call

バイトコード命令	動的束縛	メソッドの種類
invokevirtual	YES	インスタンスメソッド
invokeinterface	YES	インタフェースメソッド
invokestatic	NO	クラスメソッド
invokespecial	NO	コンストラクタ等

内でキャッチされない場合，呼び出し元メソッドへの制御フローが生じる．このメカニズムを中間表現で実現する手法として，例外の発生状態をメソッドの引数等で表現する，2 返戻値法[9)10)11)8)] が知られている．これを用いて，メソッド呼び出しを図 4 のように変換する．本稿での提案点として，関数呼び出しの直前において，例外用の戻り値として用いる引数に対し，定数 false の代入を行っている．これは，メソッドから復帰後に例外の発生状態をチェックする if 節に，デッドコード削除を適用するための細工である．呼び出し先のメソッド内において変数 exc.occured に対する代入 (KILL) が行われなければ，その後の if 文において false が定数伝搬される．よって，単純な def-use 解析と定数伝搬，デッドコード削除によって，メソッド呼び出し間の例外チェックの除去が達成される．これにより，メソッド呼び出しを越えた粗粒度並列性の抽出を，簡便に可能とした．

3. メソッド呼び出しの解析

本章では，実行時のメソッド呼び出しメカニズムを，中間表現でのプリミティブな条件分岐としてインライン展開することで，コールグラフ生成を簡略化する手法を提案する．

3.1 メソッド呼び出しが並列化に与える影響

プログラムから複数粒度の並列性を静的に抽出する上では，データ依存解析やコスト計算のために，静的なコールグラフの生成が必要である．ところが，Java はポリモーフィズムの概念を有し，表 1 の通り，呼び出すメソッドを実行時に決定する動的束縛の仕組みを持つ．動的束縛は静的なコールグラフの作成を困難にし，並列化の障害となる．

3.2 動的束縛の解析と実装の先行研究

Smalltalk のような動的型付け言語では，動的束縛は名前による lookup によって行われる．これは大きなオーバーヘッドとなるため，インラインキャッシュによる高速化の研究が行われてきた[12)13)]．

一方，Java や C++のような静的型付け言語では，動的束縛は仮想関数テーブル (vtable) と呼ばれる関数ポインタの配列によって実現されている．vtable は名前による lookup よりも高速であるが，無視できないオーバーヘッドを伴うことが知られている[14)]．このため，型推論によって動的束縛を静的束縛に変換する，非仮想化 (devirtualization) の研究が行われてきた[15)16)17)]．また，SmallEiffel において，vtable をクラス ID による二分探索に置き換えて高速化する，Binary tree dispatch (BTD) が提案されている[18)]．

```
switch(obj.classId){
case CLASS_A:
 return static_A_method(obj);
case CLASS_B:
 return static_B_method(obj);
...
```

図 5　比較分岐によるメソッドディスパッチ
Fig. 5　method dispatch by compare-branch

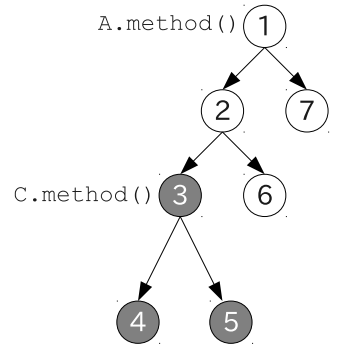

図 6　クラス継承ツリーとクラス ID
Fig. 6　class inheritance tree and class ID

加えて，ポインタエイリアス解析の研究によって，オブジェクトを細かく特定することが可能となりつつある[19)]．これを用いてオブジェクトの型をコンパイル時に確定し，メソッド呼び出しの非仮想化を行うことが可能である．

これらの型推論やポインタエイリアス解析を用いても，なお非仮想化できないメソッド呼び出しが存在する場合，コンパイラはメソッドの候補の集合を用いて，近似されたコールグラフを作成する必要がある．

3.3 コールグラフ生成を簡略化する提案手法

呼び出し箇所 (polymorphic site) ごとに異なるメソッドの候補の集合を保持して，コールグラフ生成やコスト計算，スケジューリングなどを行うことは，煩雑である．

提案手法は，メソッドの動的ディスパッチのメカニズムを，中間表現におけるプリミティブな条件分岐にインライン展開することで，ディスパッチ先のメソッドの候補が複数ある場合でも，簡便にコールグラフの生成を可能とするものである．

条件分岐への変換には BTD を用い，クラス ID による二分探索を行う．二分探索のコードが長いため，ここではより単純化したモデルとして図 5 のような switch 文を考え，メソッ

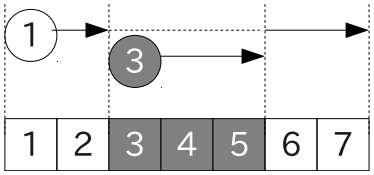

図 **7** クラス ID によるメソッド継承範囲
Fig. 7 method inheritance interval by class ID

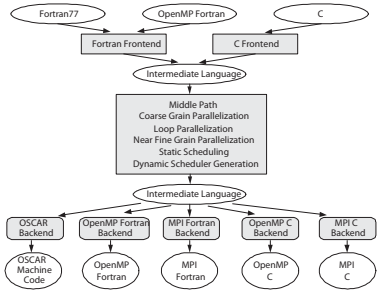

図 **8** OSCAR コンパイラの構成
Fig. 8 configuration of OSCAR compiler

ドディスパッチが比較分岐で表現できることを示す．このような比較分岐を，多態性のあるメソッドごとにディスパッチ関数として作成する．

提案手法での BTD に対する改良点として，Java において多重継承がないことを用い，インスタンスメソッドのディスパッチに限って，比較分岐回数を削減を行った．分岐数を減らすことは，コードサイズを削減し命令キャッシュを有効活用するほか，プロセッサによる分岐予測ミスを低減するため，高速化につながる．手順を以下に示す．

まず，クラスツリーを構築し，深さ優先探索を行って，訪問順にインクリメンタルな整数クラス ID を割り当てる．図 6 において，クラス A を ID1，クラス B を ID2，クラス C を ID3 とする．このとき，あるクラスの継承範囲を，そのクラス ID から，継承クラスの最大クラス ID までの整数区間で表現できることがわかる．例えば，図 6 で，クラス B における継承クラスの最大クラス ID は 6 である．すると，継承クラスの最大クラス ID を用いて，動的ディスパッチにおける比較分岐の回数を減らすことが可能である．図 7 は，クラス A において定義され，クラス C において再定義されたメソッドについて，ディスパッチ先を 3 つの整数区間で表現している．これにより，探索すべき項目数を 7 つから 3 つに減らすことができる．

4. 評　　価

本章では，2 章で述べた例外フロー除去の手法と，3 章で述べたコールグラフ生成簡略化の手法について，それぞれ評価を示す．評価マシンは IBM p5 550Q Power5+ 1.5GHz である．

表 **2** 物体追跡における配列アクセスの例外チェックの除去結果
Table 2 result of exception elimination for array access on optical flow

例外クラス	発生しうる箇所	除去できた数
ArrayIndexOutOfBoundsException	14	11
NullPointerException	14	14

表 **3** ノイズ除去における配列アクセスの例外チェックの除去結果
Table 3 result of exception elimination for array access on median filter

例外クラス	発生しうる箇所	除去できた数
ArrayIndexOutOfBoundsException	10	10
NullPointerException	10	10

表 **4** 輪郭抽出における配列アクセスの例外チェックの除去結果
Table 4 result of exception elimination for array access on gradient filter

例外クラス	発生しうる箇所	除去できた数
ArrayIndexOutOfBoundsException	5	5
NullPointerException	5	5

4.1 例外フローの除去による並列性抽出の評価

2 章で述べた提案手法である，例外フローの除去による並列性抽出を，OSCAR 自動並列化コンパイラに実装し，評価プログラムに適用した．評価時点では Java 仮想マシンへの組み込みが完了していないため，Java バイトコードから OpenMP C ソースコードへのトランスレートを行った．評価プログラムには，組み込み環境における画像処理を想定し，optical flow(物体追跡)，median filter(ノイズ除去)，gradient filter(輪郭抽出) を選択した．なお，ガベージコレクションは行われない．

配列アクセスの例外チェックについて，その除去結果を表 2 ，表 3，表 4 に示す．また，メソッド呼び出し間の例外チェックについて，その除去結果を表 5 に示す (java.lang.Math クラスのメソッドの呼び出しを除く)．さらに，並列化されたプログラムの，逐次実行に対する速度向上率を図 9，図 10，図 11 に示す．

考察を述べる．いずれのプログラムも，最外側ループで do-all 並列化された．1 プロセッサに対する 8 プロセッサでの速度向上率として，optical flow(物体追跡) で 7.92 倍，median filter(ノイズ除去) で 7.84 倍，輪郭抽出 (gradient filter) で 7.96 倍の性能が得られた．

optical flow において除去できなかった例外チェックは，いずれも，並列性抽出に影響しない逐次区間に位置するものであった．optical flow には結果を書き出すための逐次ループ

表 **5** メドッソ呼び出し間の例外除去結果
Table 5 result of inter-method exception elimination

プログラム	メソッド呼び出しの数	例外処理を除去できた数
optical flow	2	0
median filter	1	1
gradient filter	1	1

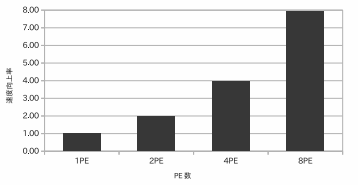

図 **9** 物体追跡を並列化した際の速度向上率
Fig. 9 speedup ratio of parallelized optical flow

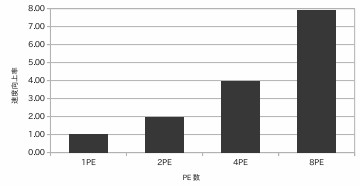

図 **10** ノイズ除去を並列化した際の速度向上率
Fig. 10 speedup ratio of parallelized median filter

が存在するにも関わらず，do-all 並列化されたループの実行時間が支配的となったため，スケーラブルな速度向上を得られた．

4.2 比較分岐によるメソッドディスパッチの評価

2 章で述べた提案手法である，コールグラフ生成の簡略化を適用すると，メソッドの動的ディスパッチが，vtable による間接参照ではなく，クラス ID による比較分岐で実行される．ここで，比較分岐による動的ディスパッチの速度が，vtable を用いた場合と比べて，少なくとも遅くならないことを確かめる必要がある．このため，マイクロベンチマークを作成し，評価を行った．

作成したマイクロベンチマークは，連続領域へのメモリアクセスを行った直後に動的ディスパッチを模した関数呼び出しを行うものである．連続領域へのメモリアクセスを行うことにより，メモリインテンシブな状況を作成した．

ディスパッチ先は a,b,c の 3 つの関数で，クラスは A,B,C,D の中から固定 (Fixed)，ローテート (Rotate)，ランダム (Random) の 3 つのパターンで選択する．クラス C とクラス D は同じディスパッチ先を持つが，比較分岐は 4 つとした．これを 5 万回繰り返したときの時間を計測し，vtable による動的ディスパッチの時間と，比較分岐による動的ディスパッチの時間の差を求めた．結果を図 12 に示す．このグラフは比較分岐と vtable を比較し，比較分岐の性能向上を示したものである．

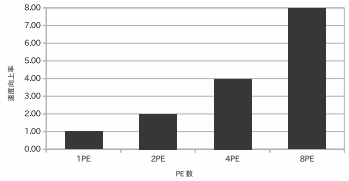

図 **11** 輪郭抽出を並列化した際の速度向上率
Fig. 11 speedup ratio of parallelized gradient fliter

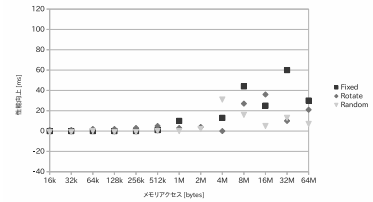

図 **12** 比較分岐と vtable でのディスパッチ速度比較
Fig. 12 speed comparison between compare-branch and vtable

考察を述べる．メモリアクセスが 16KB から 512KB までの場合，比較分岐と vtable で大きな差が見られなかった．一方，メモリアクセスが 1MB を越えるメモリインテンシブな状況下では，比較分岐の方が高速であるという知見が得られた．これは，vtable がデータキャッシュ上からパージされると，メモリアクセスによる遅延が発生することに起因すると考えられる．これらのことから，メソッド呼び出しの動的ディスパッチを比較分岐へと変換した上で解析しても，実行時の性能に支障をきたさないと言える．

ただし，メソッドのオーバーライド回数が大きい場合，比較分岐の数が増え，分岐予測ミスの増加や命令キャッシュヒット率の低下により，性能低下を引き起こすことが想像される．実際の組み込みプラットフォームにおける Java アプリケーションで，メソッドのディスパッチ先の候補がどの程度の数になるのか，調査が必要である．また，提案手法において比較分岐数を減らす手順がどの程度有効に機能するのかも，評価が必要である．

5. ま と め

本稿では Java プログラムを自動並列化するための，例外フローの除去手法，および，コールグラフ生成の簡略化手法を提案した．例外フローの除去手法を OSCAR 自動並列化コンパイラに実装し，optical flow など 3 つの画像処理プログラムを並列化したところ，IBM Power5+ 8 プロセッサにおける 1 プロセッサに対する速度向上率として，最低 7.84 倍の性能向上が得られた．また，コールグラフ生成の簡略化手法を適用しても，実行時の動的ディスパッチの性能が低下しないことを示した．

今後の課題として，現状では Java バイトコードから OpenMP C ソースコードへのトランスレートを行っている点について，Java 実行環境での JIT コンパイルを達成することが

挙げられる．

参 考 文 献

1) Artigas, P.V., Gupta, M., Midkiff, S.P. and Moreira, J.E.: Automatic loop transformations and parallelization for Java, *Proceedings of the 14th international conference on Supercomputing*, ICS '00, New York, NY, USA, ACM, pp.1–10 (2000).
2) 本多弘樹，岩田雅彦，笠原博徳：Fortran プログラム粗粒度タスク間の並列性検出手法，電子情報通信学会論文誌. D-1, 情報・システム. 1, 情報処理， Vol.73, No.12, pp. p951–960 (1990-12).
3) 小幡元樹，白子 準，神長浩気，石坂一久，笠原博徳：マルチグレイン並列処理のための階層的並列性制御手法，情報処理学会論文誌， Vol. 44, No. 4, pp. 1044–1055 (2003-04-15).
4) Suzuki, N. and Ishihata, K.: Implementation of an array bound checker, *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, New York, NY, USA, ACM, pp.132–143 (1977).
5) Kolte, P. and Wolfe, M.: Elimination of redundant array subscript range checks, *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, New York, NY, USA, ACM, pp.270–278 (1995).
6) Gupta, R.: A fresh look at optimizing array bound checking, *SIGPLAN Not.*, Vol.25, pp.272–282 (1990).
7) Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K. and Cox, D.: Design of the Java HotSpot™ client compiler for Java 6, *ACM Trans. Archit. Code Optim.*, Vol.5, pp.7:1–7:32 (2008).
8) Prabhu, P., Maeda, N. and Balakrishnan, G.: Interprocedural exception analysis for C++, *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, Berlin, Heidelberg, Springer-Verlag, pp.583–608 (2011).
9) Krall, A. and Grafl, R.: CACAO - A 64 bit JavaVM Just-in-Time Compiler, *Concurrency: Practice and Experience*, Vol.9, ACM, pp.1017–1030 (1997).
10) 千葉雄司：組込み機器向け Java2C トランスレータにおける 2 返戻値法を使った例外処理の実現，情報処理学会論文誌. プログラミング， Vol.43, No.1, pp.85–96 (2002-01-15).
11) Jung, D.-H., Park, J.K., Bae, S.-H., Lee, J. and Moon, S.-M.: Efficient exception handling in Java bytecode-to-c ahead-of-time compiler for smbedded systems, *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, New York, NY, USA, ACM, pp.188–194 (2006).
12) Deutsch, L.P. and Schiffman, A.M.: Efficient implementation of the smalltalk-80 system, *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, New York, NY, USA, ACM, pp.297–302 (1984).
13) Hölzle, U., Chambers, C. and Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, *ECOOP'91 European Conference on Object-Oriented Programming* (America, P., ed.), Lecture Notes in Computer Science, Vol. 512, Springer Berlin / Heidelberg, pp. 21–38 (1991). 10.1007/BFb0057013.
14) Driesen, K. and Hölzle, U.: The direct cost of virtual function calls in C++, *SIGPLAN Not.*, Vol.31, pp.306–323 (1996).
15) Calder, B. and Grunwald, D.: Reducing indirect function call overhead in C++ programs, *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, New York, NY, USA, ACM, pp.397–408 (1994).
16) Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7-11, 1995* (Tokoro, M. and Pareschi, R., eds.), Lecture Notes in Computer Science, Vol.952, Springer Berlin / Heidelberg, pp.77–101 (1995).
17) Bacon, D.F. and Sweeney, P.F.: Fast static analysis of C++ virtual function calls, *SIGPLAN Not.*, Vol.31, pp.324–341 (1996).
18) Zendra, O., Colnet, D. and Collin, S.: Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler, *SIGPLAN Not.*, Vol.32, pp.125–141 (1997).
19) 間瀬正啓，村田雄太，木村啓二，笠原博徳：自動並列化のための Element-Sensitive ポインタ解析，情報処理学会論文誌. プログラミング， Vol.3, No.2, pp.36–47 (2010-03-16).