1 はじめに

Java 言語に代表される，可搬性の高いプログラム言語が近年注目されている．それらの言語では，可搬性を確保するためにバイトコードなどのコンパクトかつ機種独立なプログラム中間形式を用い，高速化のために必要な部分を実行時にネイティブコードにコンパイルし，実行速度を向上させる Just-In-Time (JIT) コンパイラを用いるのが一般的となっている．しかし，JIT コンパイラは，通常のコンパイラと比較すると，その構築に関する統合的な技術フレームワークが与えられていない．そのため，JIT 自身の可搬性の欠如，最適化に関する技術的知見の不足などの問題が指摘されている．

例えば，通常のコンパイラにおける「最適化」は，ある程度メモリなどの計算資源が確保できるという仮定の基に，なるべく速度を向上させるのが主眼である．逆に，組込み型のアプリケーションで必要な “Resource-Efficiecy” (資源効率) を優先しつつ適度に高速化するような技法はあまり用いられない．現状の多くの JIT コンパイラはそのような単純な速度重視のものが多く，組み込み機器用には必ずしも適切ではないにもかかわらず，それを変更する手立てがない．より一般的には，一つのプログラムを様々な異なる計算環境へ適合させるため，(1) 個々のアプリケーション及び計算環境に特化した最適化，(2) 計算環境とアプリケーションに応じたコンパイルコードの拡張，が必要となる．しかし，従来の JIT は一種のブラックボックスになっており，(1) に関しては汎用性のある最適化しか行わず，また，(2) に関しては言語の拡張や新規の機能に対応してコード生成の手法を変えるようなカスタム化の機能は提供していない．

この問題の原因は，JIT の構築が単純に従来のコンパイラ技術を下地として，アドホックに構築されているためと考えられる．Java の JIT の多くは C 言語やアセンブラで記述されており，再利用性，可搬性，適合性など，本来ソフトウェアが持つべき性質を兼ね備えていない．通常のコンパイラでは，Stanford 大学の SUIF プロジェクト [1] のように，高品質なコンパイラをオブジェクト指向フレームワークとして実現し，様々な種類のコンパイラ構築の研究や技術開発の礎とする試みがあるが，通常の JIT は完全なブラックボックスであり，ソースコードすら公開されていない．このような現状，JIT を含む動的コンパイル技術の研究の大いなる妨げになっていると我々は考えている．

そこで，我々は上記の問題を解決するために，自己反映計算 (リフレクション) に基づいた Open Compiler(開放型コンパイラ)[2] 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラである OpenJIT を研究・開発している．OpenJIT では，JIT コンパイラ自身が Java のクラスフレームワークとして記述されており，クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化モジュールを組み込むことを可能とする．これにより，様々な計算環境・プラットフォーム・アプリケーションに対する適合や，複雑な最適化を組み込むことも可能となる．

OpenJIT により，計算環境・プラットフォーム・アプリケーションに応じた動的なコンパイルの適合化が可能になり，様々な計算環境に対して可搬性の高いプログラムが実現できる．これに対し，通常の Java のシステムは可搬性があると言われているが，その可搬性は JVM(Java Virtual Machine) が実現する仕様の可搬性に依存している．逆に，JVM が実現していない機能は，Java で簡便に記述できない限り，非標準的な手段によって実現されるより他ない．例えば，Java のマルチスレッドを用いて並列化されたプログラムを，分散メモリ計算機で実行しようとした場合，何らかの形で分散共有メモリを Java 上で実現する必要がある．ところが，JVM はそもそも分散共有メモリを一切サポートしておらず，またユーザレベルでの分散共有メモリ実現に必要な Read/Write Barrier を挿入するようなソフトウェア上の「フック」も存在しない．結果として，JVM を改造するか，プリプロセッサを用いるなど，煩雑で可搬性を妨げる手法を用いる必要がある．これに対し，OpenJIT では，コンパイラの拡張クラスファイルとして，Read/Write Barrier をコンパイルコードで一般的に実現することにより，必要な場合に分散共有メモリのコードを出力するような可搬性の高いシステムの構築が可能となる．(このように，OpenJIT は，Java に JVM が提供しない一種のフックを入れる手段として利用できる．)

また，OpenJIT により，より効果的な最適化を，クラスに特化した形で実現することが可能となる．これにより，効果的だがコストの高い最適化を，プログラムの必要な部分だけに適用することが可能となる．たとえば，Java ではあまり行われてこなかった数値計算の最適化 (ループ変換・各種キャッシュブロッキング・データ分散のアドレス計算の最適化など) を，必要なクラスの集合に対して行うことも可能となる．このような最適化は，従来の Fortran や C のコンパイラでは必然であったが，JIT コンパイラにおいては全てのプログラムに対して行うにはコストが高すぎるとされてきた．一方，OpenJIT では，そのような最適化を，必要に応じてクラスライブラリ・アプリケーションの作成者，およびユーザが必要なクラスに対して場合に応じて行える．さらに，最適化が余りにもアプリケーションに特化しすぎており，汎用の JIT コンパイラに組み込むことができない場合にも対応が可能になる．例えば，低レベルの画像マルチメディア処理では，複数の画像処理オペレーターを合成すると数十倍から数百倍の速度向上が得られることが報告されているが，ソースレベルでそのようの合成を行うと，ソースプロ

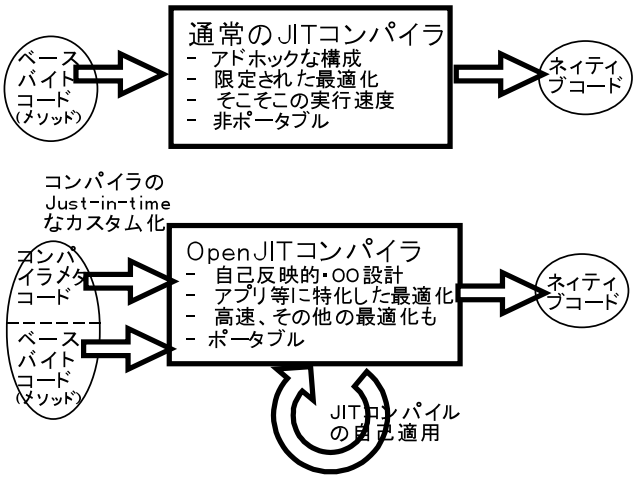

図 1 従来の JIT と OpenJIT の比較の概念図

グラムが極めて煩雑なものとなる．そこで，何らかの特化したコンパイラが望まれるが，そのような特化したコンパイラの作成は難しく，かつソースの機密性も問題となる． OpenJIT では，バイトコードレベルでそのような合成をするような特殊なコンパイルをするモジュールを組み込むことによって，対応が可能となる．

このように， OpenJIT は従来のコンパイラと比較し，新たなプログラミングスタイルを可能とする．すなわち，従来の JIT を前提にしたプログラミングでは，図 1の上図のように，ベースレベルのプログラムを JIT コンパイラがコンパイルし，プログラマやユーザは JIT コンパイラはが必要な最適化をすることを期待するか，あるいは何らかのソースレベルでの変換をし，オリジナルのプログラムと比較すると見通しの悪いプログラムをメンテナンスする他なかった．しかし， OpenJIT により，図 1の下図のように，プログラマはは見通しの良い Java のベースレベルのコードを記述し，プログラマ自身，あるいはそのプログラムのユーザが適切な最適化・適応化クラスライブラリを選択することにより，ベースレベルの記述の簡潔さを保存したまま，効率化を図ることができる．このような性質は自己反映システム一般が持ちあわせているものであるが， JIT コンパイラとして実現することにより，従来の自己反映システムと比較して，プログラムのソースをユーザに開示することなく，ポータブルに，かつ高い実行効率を得ることが可能となる．

自己反映的なコンパイラや，コンパイラフレームワークの実現は過去にも例があるが， OpenJIT は， Java の Just-In-Time コンパイラであることにより，幾つかの従来のコンパイラにはない技術的特徴を持つ．以降では OpenJIT の概要を示し，その第一プロトタイプの現状と性能評価を示す．

2 関連研究

JIT コンパイラは，古くは Lisp や Smalltalk などの，中間言語系の言語処理系を高速化させるための技術として開発された (例として， ParcPlace Smalltalk の Deutch-Schiffman の最適化技法などが挙げられる [3])．しかし，これらは Java における JIT コンパイラと同様， C 言語などの低レベルな言語で記述され，自己反映的カスタマイゼーションの機能はない．

自己反映計算は， 1982 年に Indiana 大学 (当時 MIT) の Brian Smith の提唱した 3-Lisp によって最初の理論的な基盤が提唱された．基本的には，適切な計算系の定義により，通常の計算を表す従来からのベースレベルのコードに加え，自己の計算系を表すメタレベルのコードをユーザは記述することが可能となる．Smith の研究以後，様々な研究により，プログラミング言語やオペレーティングシステム，あるいは分散システムにおいて，それらの計算 / 実行モデルを自己反映計算モデルとすることにより，従来では例外的かつ固定的な形で扱われてきた各種の機能 (例外処理，資源管理など) を，整合的かつ柔軟な方法でモデルに導入できることが示された．これらの言語や OS は，簡潔性とともに非常に高い拡張性を提供している．また，自己反映的なプログラミング技法の応用として，プログラム言語処理系のプロトタイピング，ウィンドウシステム記述や，組込み型の OS の構成，分散システムにおける数々の資源制御などが発表されている．

Open Compiler は，コンパイラのコンパイル時の挙動を自己反映的に変更する機能を提供するシステムである．コンパイラをオブジェクト指向設計に基づきモジュール化してユーザから変更可能にし，さらにコンパイル時にメタ計算をあらかじめ静的に行うことによって，言語の拡張やアプリケーション固有の最適化などを行う．古くは Lisp のマクロがアドホックに Open Compiler 的な機能を提供したが，明示的に自己反映計算系としては Open Compiler は Xerox PARC の Anibus/Intrigue[4] が先駆的であるが， MPC++[5], OpenC++ v.2.0[6], EPP[8] など，様々な処理系が提案・研究されている．また，並列オブジェクト指向言語 ABCL/R3[7] において， JavaVM のようなインタプリタによる解釈実行に基くメタレベル記述を，部分計算 (partial evaluation) によってコンパイルする技法が提案されている．また Consel らは Java において部分計算を用いて高速化を図る手法を提案している [9].

OpenJIT は，今までの Open Compiler とは異なり， JIT の各モジュールを実行時に置き換え，自己適用的な最適化を行うので， ABCL/R3 と Open Compiler の中間的な存在であると言える．従って，両方の技術を基盤とし，かつ融合させることによって，高速な処理系を構築することを目指しているとも言える．また，全体的に見ると， EPP のような Open

Compiler が OpenJIT のフロントエンドとなることも可能であり，それぞれ相補的な技術であるといえる．

3 OpenJIT の技術的課題

OpenJIT は自己反映的なシステムであるため，Java 言語自身のクラスフレームワークとして構築される．これにより，以下の従来のコンパイラにはない技術的課題がある．

1．コンパイラ自身の自己反映的動的コンパイル:

OpenJIT 自身，ほとんど Java で記述される．従って，OpenJIT によるコンパイラのコードは，当初 JVM で解釈系実行され，次第に自身のネイティブコード化により高速化・最適化がなされていく．これにより，JIT コンパイラ自身をその実行環境に自動的に適合していくことが可能となるが，逆に以下のような問題が生じる可能性がある:

- JIT コンパイラの性能の問題: JIT コンパイラでは，(コンパイル時間 + ネイティブコードの実行時間) が JVM 上での解釈実行時間を上回らなくては意味がなく，現実的には従来の通常の JIT コンパイラと十分競える速度を得なくてはならない．ところが，上記のように OpenJIT は当初 JVM にて解釈実行され，ネイティブコードになった時点でも，自身のコード生成の質により必ずしも C で記述された JIT コンパイラの性能が得られるとは限らない．実際のプログラムにおいて，OpenJIT が十分な基礎的性能を得られるか否か，は実験によって検証する必要がある．
- JVM の適切な JIT コンパイラ向けの API の欠落の問題: JIT コンパイラは，コンパイル時に JVM 内の様々なデータ構造を参照，変更する必要がある．ところが，現状の JVM はそのような API を一切持たない．これは (1)JIT コンパイラが C など，低レベル言語で記述されていることが前提とされ，Java 自身で構築されることが前提とされていないため，(2) セキュリティと安全性のため，と考えられる．後者は本質的な問題であるが，なるべくそのような問題を回避しつつ，Java 言語から用いやすい適切な JVM との API を定義し，JVM とリンクする必要がある．
- 自己可変コードの問題: 通常の JIT コンパイラならばこれは問題ないが，OpenJIT は自分自身をコンパイルするために，自己可変コードとなる．一方，Java の実行系は本質的にマルチスレッド環境なので，必要なロッキングを行う必要があり，それにより race condition やデッドロックが起きる可能性がある．
  また，マルチプロセッサ環境では正しくキャッシュの一貫性を保つために，正しくフラッシュを行う必要が生じる．

2．コンパイラの実行時の変更:

OpenJIT では，コンパイラ全体に対して拡張や最適化の指示を行うモジュールを実行時に動的に組み込むことを可能とする．これらの拡張や最適化は，Java のクラスファイルに隠蔽されるため，そのクラスライブラリやそれに基づくフレームワークを用いるユーザは，自らは様々なハッキングを行うことなく，自動的にその最適化の利益を享受することができる．しかし，このような性質に関しても，問題が生じる可能性がある:

- 自己可変性の制限の問題: OpenJIT は，通常は場合によっては，コンパイラ自身はカスタム化されたコンパイルや特殊な最適化の対象としたくない場合もある．このため，適切な scope control[10] の機構が必要となる．
- コンパイラの安全性の問題: 関連するが，任意の部品が組み込める場合，コンパイラの安全性が問題となる．
- コンパイラ変更の API の問題: コンパイラが動的に可変であるためには，適切な API を定義する必要がある．特に，Java の場合は，重複したクラスの定義や再ロードはできないので，delegation を用いる，あるいは Factory Pattern を用いるなど，動的な再構成を許すようなクラス・オブジェクト間の構成にし，適切な API を設計する必要がある．

これらの問題に対し，OpenJIT では，以下のような解決策をとる (あるいはとる予定である):

1．コンパイラ自身の自己反映的動的コンパイル:

- OpenJIT の性能の問題に関しては，通常の JIT のコンパイルパスをなるべく簡潔なものとし，従来の JIT で短い最適化時間で効果のあるとされるスタックコードからレジスタコードへの変換時の工夫，レジスタ割付の最適化，Peephole 最適化などを中心に行う．逆に，時間がかかるメソッド間最適化などは，デフォルトの OpenJIT のコンパイルパスでは行わない．それでも C の JIT と比較すると速度低下が起こる可能性があるが，我々のプロトタイプによる実験では，C の JIT と OpenJIT でほとんど性能差がないことが判明している．この理由は，主に JIT の実行時間が，そもそも全体の実行時間内に占める割合が小さい (文献 [11] によると，10% 以下) ことに起因する．
- JVM の適切な JIT コンパイラ向けの API の欠落の問題に対しては，C で JVM 内部のアクセスを行う Native Method のスタブを作成し，

OpenJIT はそれを呼ぶようにする．このためのインターフェースは，そもそも通常の JIT コンパイラに対しては開放されていたが，Java に対する API はなかった．OpenJIT では，一連の JIT 向けの API をクラスとして定義し，JVM に対しては C のスタブライブラリが JIT コンパイラに見えるようにして，Java に対して JVM 内部を開放している．(本来は，安全性と標準化のために，Sun がこのような API を準備すべきである．)

- 自己可変コードの問題: OpenJIT では，デッドロックを避けるため，自己可変コードの部分はロックフリーなアルゴリズムを用いるようにしている．また，適切にコードキャッシュを選択的にフラッシュすることにより，実行効率の低下を避けている．

2．コンパイラの実行時の変更:

- 自己可変性に関しては，ある JIT コンパイラに対するカスタム化のクラスライブラリが，それを用いるコンパイル時に，(1) OpenJIT 自身と，それ以外のユーザのクラス全体のコンパイルに適用される場合，(2) OpenJIT 以外の全てのユーザのクラス全体のコンパイルに適用される場合，(3) ある特定のクラスのコンパイルにのみ適用される場合，の 3 通りの scope control を可能とする．
- 安全性に関しては，上記の scope control によってある程度保証することは可能であるが，コンパイラのカスタム化によって安全性の低いコードが混入する可能性があることは否めない．これに関しては，現状では手だてはなく，JVM 側での Protection Domain などの導入，および安全とみなされたカスタム化ライブラリに対する authentication などの手段をとる必要がある．
- コンパイラ変更の API の問題: 現状のプロトタイプの OpenJIT では，比較的大きいクラス単位の機能分割がしてあり，必ずしも動的な変更に適していない．これに関しては，変更に適したデザインパターンを用いて幾度かの design iteration により refine していく必要があり，現在もっとも大きな技術的課題である．

4 OpenJIT の機能構成の概要

OpenJIT は，Java 言語自身でポータブルに記述され，Java Virtual Machine の JIT に対する標準 API を満たすように作成される．

OpenJIT の機能は，大きく分けてフロントエンド系の処理とバックエンド系の処理の 2 つから構成される．フロントエンド系は，Java のバイトコードを入力とし，高レベルな最適化を施して，バックエンドに渡す内部的中間表現，あるいは再びバイトコードを出力する．バックエンド系は，得られた中間表現あるいはバイトコードに対し，より細かいレベルでの最適化を行い，ネイティブコードを出力する．フロントエンド系とバックエンド系間のインターフェースに中間表現とバイトコードの両者を用意したのは，システム全体の処理性能を重視するとともにモジュラリティを確保するためである．

以下ではフロントエンド系，バックエンド系の処理に関して詳細に述べる．

4.1 フロントエンド系

OpenJIT の起動は，JDK の提供する JIT インターフェースを介してメソッド起動ごとに行われる．OpenJIT が与えられたメソッドに対して起動されると，フロントエンド系は対象となるバイトコード列に対して以下の処理を行う．

まず，ディスコンパイラモジュールがバイトコードから AST を逆変換により得る．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフの復元を行う．同時に，アノテーション解析モジュールによって，対象となるクラスファイルに何らかのアノテーション情報が付記されていた場合にその情報を得る．この情報は，AST 上の付加情報として用いられる．

次に得られた AST に対し，最適化モジュールによって最適化が施される．最適化は，通常の AST とフローグラフを用いた構築・解析・変換を行い，最適化モジュールのサポートモジュールであるフローグラフ構築モジュール，フローグラフ解析モジュール，プログラム変換モジュールを用いて実現される．フローグラフ構築モジュールは対応するデータ依存グラフ，コントロール依存グラフ等を生成する．フローグラフ解析モジュールはこれらのデータフローグラフに対するデータフロー問題，マージ，浮動点検出等の諸ルーチンをメソッドとして提供する．プログラム変換モジュールは解析情報を元にテンプレートマッチングによって AST 上の変換を行う．

変換後の結果はバックエンドが必要とする中間表現ないしバイトコードに変換されて，出力される．

フロントエンド系の処理の概略を図 2に示す．各モジュールの機能は以下に示す通りである．

1. OpenJIT コンパイラ基盤モジュール: OpenJIT 全体の基本動作を司る．
2. OpenJIT バイトコードディスコンパイラモジュール: Java のクラスファイルに含まれるバイトコードを，いわゆる discompiler 技術により，コンパイラ向けの中間表現に変換する．
3. OpenJIT クラスファイルアノテーション解析モジュール: プログラムグラフ (AST) に対して，

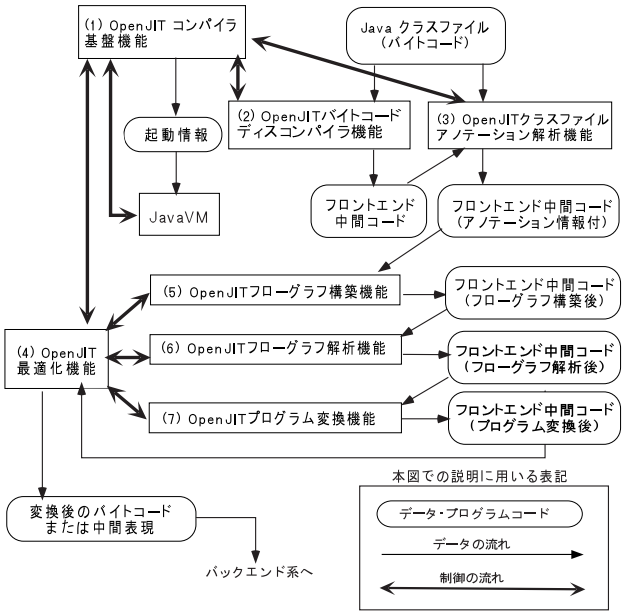

図 2 OpenJIT フロントエンド系の概要

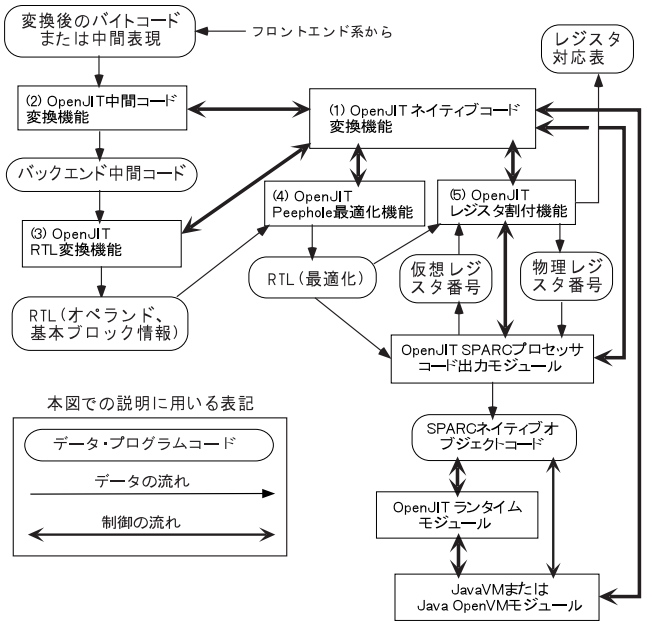

図 3 OpenJIT バックエンド系の概要

コンパイル時に適切な拡張された OpenJIT のメタクラスを起動できるようにする．

4. OpenJIT 最適化モジュール: 各種解析モジュールおよびプログラム変換モジュールを用い，プログラム最適化を行う．
5. OpenJIT フローグラフ構築モジュール: AST およびコントロールフローグラフを受け取り，対応するデータ依存グラフ，コントロール依存グラフ，などのフローグラフを出力する．
6. OpenJIT フローグラフ解析モジュール: フローグラフ構築モジュールで構築されたプログラム表現のグラフに対し，グラフ上の解析を行う．
7. OpenJIT プログラム変換モジュール: 解析モジュールの結果やユーザのコンパイラのカスタマイゼーションに従って，プログラム変換を行う．

4.2 バックエンド系

バックエンド系はフロントエンド系によって出力された中間表現ないしバイトコードに対して，低レベルの最適化処理を行い，ネイティブコードを出力する．ネイティブコード変換モジュールはバックエンド系処理全体の抽象フレームワークであり，バックエンド系のモジュールのインタフェースを定義する．このインタフェースに沿って具体的なプロセッサに応じたクラスでモジュールを記述することにより，様々なプロセッサに対応することが可能となる．

まず，中間コード変換モジュールによって，バイトコードの命令を解析して分類することにより，スタックオペランドを使った中間言語の命令列へと変換を行う．フロントエンド系が中間表現を出力する場合はこの形式で出力するため，この変換処理は省略される．次に RTL 変換モジュールは，得られた命令列に対し，スタックオペランドを使った中間言語から仮想的なレジスタを使った中間言語 (RTL) へ変換する．この際，バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．また，バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．

次に，Peephole 最適化モジュールによって，RTL の命令列の中から冗長な命令を取り除く最適化を行い，最適化された RTL が出力される．最後に OpenJIT SPARC プロセッサコード出力モジュールにより，SPARC プロセッサのネイティブコードが出力される．出力されたネイティブコードは，JavaVM によって呼び出され実行されるが，その際に OpenJIT ランタイムモジュールを補助的に呼び出す．

バックエンド系の処理の概略を図 3に示す．各モジュールの機能は以下に示す通りである．

1. OpenJIT ネイティブコード変換モジュール: OpenJIT バックエンド全体の制御を行う．
2. OpenJIT 中間言語変換モジュール: バイトコードから中間言語への変換を行う．
3. OpenJIT RTL 変換モジュール: 中間言語から RTL への変換を行う．
4. OpenJIT Peephole 最適化モジュール: RTL を最適化し，無駄な命令を取り除いて最適化を行う．
5. OpenJIT レジスタ割付モジュール仮想レジスタから物理レジスタへの割付を行う．

6. OpenJIT Sparc プロセッサコード出力モジュール: RTL からネイティブコードへの変換を行う.
7. OpenJIT ランタイムモジュールネイティブコードが実行時に呼び出し，ネイティブコードの実行ををサポートする.

5 OpenJIT の現状と性能評価

現状では，OpenJIT のバックエンド系のプロトタイプが完成している．ベースとなったのは富士通が作成した C の JIT コンパイラ (FJIT) であり，それをオブジェクト指向風に変換しつつ，Java に移植を行った．このプロトタイプは小さい (Java で約 5000 行) が，十分堅固でかつ後述するように高速であり，CaffeineMark などのマイクロベンチマーク，javac などの比較的大きなツール群，そして HotJava までが動作している．OpenJIT を動作させるには JDK 1.1.X の環境でインストールした後，以下の環境変数を設定する (<dir> は OpenJIT を展開した directory).

```
% setenv JAVA_COMPILER OpenJIT
% setenv CLASSPATH <dir>
% setenv LD_LIBRARY_PATH <dir>
```

これで，OpenJIT のクラスファイル，および C のスタブルーチンとランタイムルーチン (libOpenJIT.so) がロード可能なパスに入る．あとは通常の Java のプログラムを起動すれば，OpenJIT が自動的に起動され，通常の JIT と同様，実行時にネイティブコードに変換され，高速化される．現状では scope control の機能がないため，実質的には OpenJIT のクラスファイルが当初ほぼ占有的にコンパイルされ，その後ユーザのバイトコードがコンパイルされる.

現状の OpenJIT のバックエンドに対して，複数の性能評価を行った．全てのベンチマークは，Sun UltraSparc-II 247Mhz, Solaris 2.6, JDK 1.1.6 の環境で行った.

CaffeineMark 3.0:

CaffeineMark 3.0[12] は，Pendragon Software 社の提唱している Java のベンチマークである．内容としては Sieve, Loop, Logic, Method, Float, Graphics, Image, Dialog という数種類のプログラムを，それぞれ同じ時間繰り返し走らせ，動作回数がスコアとなる．ベンチマークのサイズ的にはマイクロベンチマークであると言え．正確なベンチマークとしての評価は決して高くはないが，多くの Java の VM や JIT コンパイラがベンチマーク指標として用いている.

図 4のグラフは，それぞれ JDK1.1.6 の JVM でのインタプリタ，その JVM 上で Sun の JIT，富士通の FJIT, および OpenJIT を動作させたときの CaffeineMark 3.0 の Embeded 版のスコアを表している．図で見るように，Logic を除き，FJIT と OpenJIT はごく近い性能を示しており，いずれも JVM のインタプリタより圧倒的に高速である．原因としては，CaffeineMark では比較的 JIT のオーバーヘッドが顕在化しにくいので，ほぼ共通のネイティブコードを出す FJIT と OpenJIT の性能が似通った，とも言える.

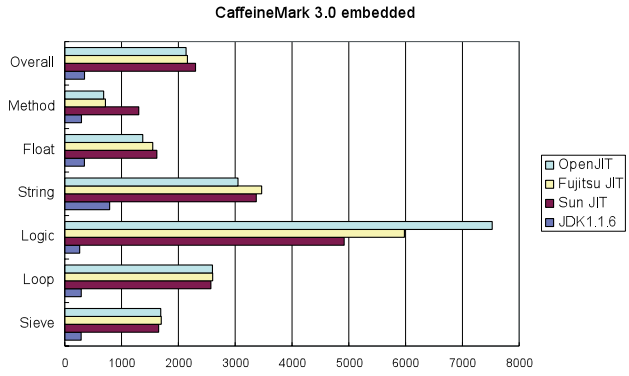

図 4 OpenJIT の CaffineMark 3.0 ベンチマーク

Javac:

CaffeineMark と比較して，より現実的なベンチマークとして用いられるのが javac である．これは javac を javac で自己コンパイルをして，その総計時間を計るものである．Javac は I/O が多く，Native method 呼び出しが多いため，JIT の効果が比較的低いことが知られている.

図 5のグラフは，JDK1.1.6 の JVM でのインタプリタ，SunJIT，FJIT, および OpenJIT の実行時間を表している．JIT はそれぞれインタプリタより倍近く速く，それぞれの JIT 同士の性能差はわずかである．これにより，OpenJIT は現実的なプログラムにおいて，C 言語で記述された JIT と比較しても十分性能を確保できていると言える.

Null Program

最後に，OpenJIT の起動時間を計測する．先に述べたように，OpenJIT は起動時には自動的にほぼ占有的に自分自身をコンパイルするため，その起動オーバーヘッドが問題となる．そこで，main のみで構成される空の “Null Program” をコンパイルし，JDK でのインタプリタ実行と比較する．具体的なプログラムは，以下の通りである:

```
class empty {
  public static void main(String argv[]) {}
}
```

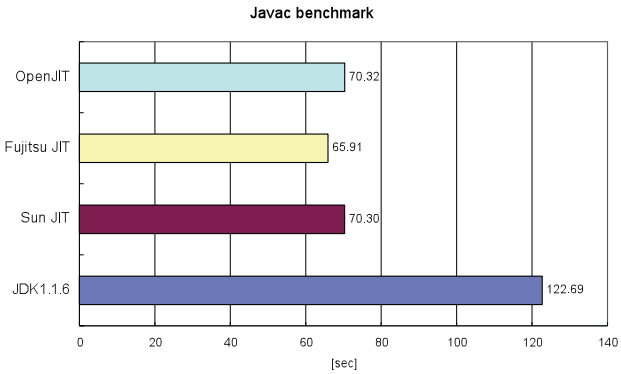

図 5 OpenJIT の Javac ベンチマーク

表 1 Null Program による OpenJIT の起動時間の計測

	OpenJIT(秒)	JDK Int. (秒)	差 (秒)
real	1.138	0.346	0.79
user	0.880	0.142	0.74
sys	0.117	0.075	0.042

結果を表1に示す．これによると， OpenJIT の起動オーバーヘッドは 1 秒以内である[2]．また，その差はほぼコンパイルに費やしていると思われる User 時間によるものであり， System 時間は大差はない．

以上のベンチマークにより，実用的な大きさのプログラムでは， OpenJIT のプロトタイプバックエンドは従来の JIT と遜色ない性能を示していると言える．

一方，まだ OpenJIT のフロントエンドはデザイン段階であり，バックエンドのクラス構造もカスタム化が可能なように様々なデザインパターンを用いて再設計する予定である．特に，フロントエンドにおける最適化は，通常のスタティックコンパイラにおける最適化とほぼ同等であり，高レベルな最適化を実行時間とのトレードオフを鑑みながら適用することになる．

6 おわりに

本稿では，自己反映計算に基づく Java の JIT コンパイラである OpenJIT の概要を紹介した． OpenJIT はまだ開発段階にあるが，既にプロトタイプは期待以上の性能を示しており，かつコンパクトでコードも大変分かりやすい．今後クラスフレームワークの洗練されたデザインを行い， API を整備することによって， JIT コンパイラの研究の礎となり，かつ実用的にも様々な特殊化された適合化，最適化が可能なシステムにしていきたい．完成の暁には， OpenJIT は PDS として無料配布される予定である．今後の詳細は `http://openjit.is.titech.ac.jp` を参照されたい．

[2] Null Method では何もコンパイルされないように見えるが，実際は初期化の時点で様々なコンストラクタのコードを実行されるので，コンパイラはほぼ全てネイティブコードに変換される．

謝辞

なお，本研究は情報処理振興事業協会 (IPA) の高度情報化支援ソフトウェア育成事業において，テーマ名「自己反映計算に基づく Java 言語用の開放型 Just-in-Time コンパイラ OpenJIT の研究開発」として実施された．

参考文献

[1] The Stanford SUIF Compiler Group: SUIF Compiler System, `http://suif.stanford.edu`

[2] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf: An architecture for an open compiler, In *Proceedings of IMSA'92: Reflection and Meta-Level Architecture*, pp. 95–106, Tokyo, Nov. 1992.

[3] L. Peter Deutsch and Alan Schiffman: Efficient Implementation of the Smalltalk-80 System, Proceedings of POPL'84, 1984.

[4] Luis Rodriguez, Jr.: A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler, In *Proceedings of IMSA'92: Reflection and Meta-Level Architecture*, pp. 107–112, Tokyo, Nov. 1992.

[5] Yutaka Ishikawa et. al.: Design and Implementation of Metalevel Architecture in C++ –MPC++ Approach–. In *Proceedings of Reflection'96*, pp.141–154, San Francisco, 1996.

[6] Shigeru Chiba: A Metaobject Protocol for C++. In *Proceedings of ACM OOPSLA'95*, pp. 285–299., Oct. 1995.

[7] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages using Partial Evaluation. In *Proceedings of ACM OOPSLA'95*, pp. 300–315., Oct. 1995.

[8] Yuuji Ichisugi and Yves Roudier: The Extensible Java Preprocessor Kit and a Tiny Data Parallel Java. In *Proceedings of ISCOPE'97*, Springer LNCS 1343, pp. 153–160, Marina Del Rey, CA, 1997.

[9] E. Volarschi, C. Consel, and C. Cowan: Declarative Specialization of Object-Oriented Programs. In *Proc. ACM OOPSLA*, pp. 286-300, 1997.

[10] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. "The Art of the Metaobject Protocol", The MIT Press, Cambridge, MA, 1991.

[11] 志村 浩也, 木村 康則: Java JIT コンパイラの試作, 情報処理学会研究報告 96-ARC-120, pp.37–42, Dec., 1996.

[12] Pendragon Software: The CaffeineMark 3.0 Benchmark, `http://www.pendragon-software.com/pendragon/cm3/index.html`.