LLVM/clang 上の CbC コンパイラの改良と Gears OS サポート

135756F 氏名 宮城光希　指導教員：河野 真治

1　研究目的

プログラムを記述する際に通常の処理の他に、メモリ管理、スレッドの待ち合わせやネットワークの管理、エラーハンドリング等、記述しなければならない処理が存在する。これらの計算を Meta Computation と呼ぶ。

Meta Computation を通常の計算から切り離して記述するためには処理を細かく分割する必要がある。しかし、関数やクラスなどの単位は容易に分割できない。

そこで当研究室では Meta Computation を柔軟に記述するためのプログラミング言語の単位として Code Segment、Data Segment という単位を提案している。

Code Segment は関数に比べて細かく分割されているので Meta Computation をより柔軟に記述できる。

Code Segment、Data Segment にはそれぞれメタレベルの単位である Meta Code Segment、Meta Data Segment が存在し、これらを用いて Meta Computation を実現する。

また、処理を Code Segment、データを Data Segment とした単位を用いたプログラミング言語 Continuation based C (CbC) を開発している。

CbC は軽量継続による遷移を行うので、継続前の Code Segment に戻ることはなく、状態遷移ベースのプログラミングに適している。

CbC と同様に処理を Code Gear、データを Data Gear とした単位を用いた並列実行をサポートした Gears OS も開発している。 Gears OS は CbC で記述されている。

本研究では、LLVM/clang 上での CbC コンパイラの改良と Gears OS の構文のサポートを行う。

2　Continuation based C (CbC)

CbC では C の関数の代わりに Code Segment を用いて処理を記述している。

Code Segment の宣言は C の関数の構文と同様に記述し、型に __code を使うことで宣言できる。

Code Segment は C の関数と異なり戻り値を持たず、Code Segment 間の移動は goto の後に Code Segment 名と引数を並べて記述する独自の構文を用いて行う。

この goto による処理の遷移を継続と呼ぶ。図 1 は Code Segment 間の処理の流れを表している。

C では関数呼び出しを行う際、現在までの位置を環境として保持するが、戻り値を持たない Code Segment は呼び出し元の環境を保持しない。ここで環境とは現在のスタックの状態のことを指す。このように呼び出し元の環境を持たない継続を軽量継続と呼ぶ。軽量継続によって、並列化、ループ制御、関数コールといったスタックを意識した最適化がソースコードレベルで行うことができる。

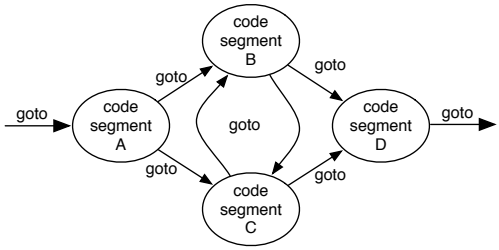

図 1: goto による Code Segment 間の接続

しかし CbC と C の記述を交える際、CbC の code segment から C の関数への呼び出しは問題なく行えるが、逆の場合呼び出し元の環境に戻るための特殊な継続が必要となる。これを環境付き継続とよぶ。

環境付き継続を用いる場合 C の関数から code segment へ接続する際に __return、__environment という変数を渡す。

3　LLVM/clang

LLVM とは、モジュラー構成および再利用可能なコンパイラとツールチェーン技術等を開発するプロジェクトの名称である。単に LLVM といった場合は LLVM Core のことを指す。以降は本文でも単に LLVM といった場合は LLVM Core のことを指す。

LLVM はコンパイラ基盤であり、コンパイラのバックエンド部分の処理を行う。LLVM IR や LLVM BitCode と呼ばれる独自の中間言語を持ち、それを機械語や実行可能なファイルに変換することができる。また、この言語で書かれたプログラムを実行するための仮想機械としても動作する。

LLVM のフロントエンドはターゲットの言語を LLVM IR に変換することによって LLVM の最適化機構を利用することができる。

clang は LLVM をバックエンドとして利用する C/C++/Objective-C のコンパイラである。

フロントエンドとして clang を使用し、与えられたコードから LLVM IR を生成し、LLVM はこれを最適化し機械語に変換を行う。

LLVM は LLVM IR を直接的にターゲットのアセンブリ言語へ変換を行うわけではなく、中間表現を何度か変えその度に最適化を行い、最終的にターゲットのアセンブリ言語に変換する。

LLVM では、最適化や中間表現の変換といったコンパイラを構成する処理は全て pass が行う。多くの pass は最適化のために存在し、この pass を組み合わせることにより、LLVM の持つ機能の中から任意のものを利用することができる。

LLVM ターゲットのアセンブリ言語を生成するまでの過程を記すと以下のようになる。

SelectionDAG Instruction Selection (SelectionDAGISel)

LLVM IR を SelectionDAG (DAG は Directed Acycric Graph の意) に変換し、最適化を行う。その後 Machine Code を生成する。

SSA-based Machine Code Optimizations

SSA-based Machine Code に対する最適化を行う。各最適化はそれぞれ独立した pass になっている。

Register Allocation

仮装レジスタから物理レジスタへの割り当てを行う。ここで PHI 命令が削除され、SSA-based でなくなる。

Prolog/Epilog Code Insertion

Prolog/Epilog Code の挿入を行う。どちらも関数呼び出しに関わるものであり、Prolog は関数を呼び出す際に呼び出す関数のためのスタックフレームを準備する処理、Epilog は呼び出し元の関数に戻る際に行う処理である。

Late Machine Code Optimizations

Machine Code に対してさらに最適化を行う。

Code Emission

Machine Code を MC Layer での表現に変換する。その後さらにターゲットのアセンブリ言語へ変換し、その出力を行う。

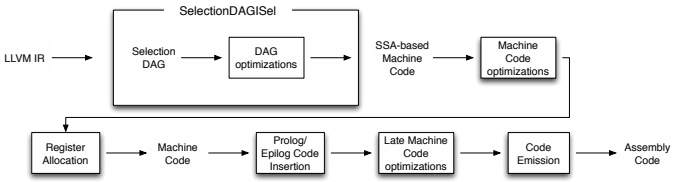

図 2: LLVM の 処理過程

4 LLVM/clang のリファクタリング

LLVM/clang で CbC をコンパイルした際 Code Segment 内の局所変数でポインタを参照すると tail call されないという不具合があることがわかった。

局所変数でポインタを参照していると clang は生成する LLVM IR に オブジェクトの寿命を示す lifetime.start と lifetime.end を書き出す。

ここでオブジェクトの lifetime の終わりを示す lifetime.end が tail call の後に書き出されてしまうことにより、tail call の際に局所変数が解放されておらず lifetime が残っているので tail call が無視されてしまう。

しかし CbC では継続を行った後、継続前の Code Segment に戻ることはないので局所変数の解放は継続前に行っても良い。そこで lifetime.end を tail call の直前で生成を行うことで tail call を出すようにした。

5 Gears OS の構文サポート

Gears OS では並列実行するための Task を実行する Code Gear と実行に必要な Input Data Gear 、出力される Output Data Gear の組で表現する。Input/Output Data Gear によって依存関係が決定し並列実行を行う。

Gears OS では Meta Computation を Meta Code Gear、Meta Data Gear で表現する。Meta Code Gear は通常の Code Gear の直後に遷移され、Meta Computation を実行する。

CbC では処理を Code Segment を用いたプログラミング言語であるため、 Gears OS の Code Gear を記述するのに適している。そこで Gears OS の構文のサポートを CbC を用いて行う。

Gears OS では Context という接続可能な Code/Data Gear のリスト、TaskQueue へのポインタ、Persistent Data Tree へのポインタ、Temporal Dara Gear である。Gears OS では必要な Code/Data Gear に参照したい場合、Context を通す必要がある。

Context は膨大であるためプログラマが記述するには負担であるため CbC では Context の自動生成を行うようにする。

6　今後の課題

本研究では LLVM/clang によるコンパイルの際、Code Segment 内の局所変数で tail call が出ない不具合を修正した。今後の課題としては、まだ Gears OS の構文サポートの実装が行われていないのでこれを実装することが優先される。

参考文献

[1] 徳森 海斗 : LLVM Clang 上の Continuation based C コンパイラ の改良,

[2] 伊波立樹, 東恩納琢偉, 河野真治 : Code Gear、 Data Gear に基づく OS のプロトタイプ, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS)(2016).

[3] 大城 信康, 河野真治 : Continuation based C の GCC 4.6 上の実装について, 第 53 回プログラミング・シンポジウム (2012)

[4] 与儀健人, 河野真治 : Continuation based c コンパイラの gcc-4.2 による実装, 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS)(2008).

[5] The LLVM Compiler Infrastructure

[6] LLVM Language Reference Manual