2015年9月25日

株式会社トランス・ニュー・テクノロジー

「Omni XcalableMPコンパイラの  
C/C++フロントエンドの開発」報告書

目次

[1 本報告書の構成 2](#_Toc430946048)

[2 納品ソフトウェアの利用方法 3](#_Toc430946049)

[2.1 gitリポジトリの構成 3](#_Toc430946050)

[2.2 ビルド手順 3](#_Toc430946051)

[2.2.1 Clang-3.6 のインストール 3](#_Toc430946052)

[2.2.2 CtoXcodeMLのビルド 5](#_Toc430946053)

[2.3 ビルドされたツールの利用方法 5](#_Toc430946054)

[3 作業報告 7](#_Toc430946055)

[3.1 納品ソフトウェアの内部実装に関する解説 7](#_Toc430946056)

[3.1.1 各ソースコードについて 7](#_Toc430946057)

[3.1.2 実装に用いているC++デザインパターンについて 8](#_Toc430946058)

[3.2 テスト用ソースコードによる評価 11](#_Toc430946059)

[3.2.1 High Performance Linpackによる評価 12](#_Toc430946060)

[3.2.2 RandomAccessによる評価 16](#_Toc430946061)

[3.2.3 STREAMによる評価 16](#_Toc430946062)

# 本報告書の構成

本報告書は、本案件に関する解説書を兼ねた文書であり、下記の内容からなる。

* 納品されたソフトウェア（CtoXcodeML）の利用方法について  
  下記の各パートからなる。
  + gitリポジトリの構成
  + ビルド方法
  + ビルドされたツールの利用方法
* 作業報告書一式  
  下記の各パートからなる。
  + 納品ソフトウェアの内部実装に関する解説
  + テスト用ソースコードによる評価

# 納品ソフトウェアの利用方法

本案件で開発されたソフトウェアは、omni-compiler.org上に設置されたgitレポジトリでバージョン管理されており、納品物はここに置かれている。

## gitリポジトリの構成

* docs/ ディレクトリ
  + docs/CtoXcodeMLreport-20150925.docx  
    本ドキュメントである。
* src/ ディレクトリ  
  ソースコード一式を含む。

## ビルド手順

まず、Clang-3.6をソースコードからビルドし、 /usr/local 以下にインストールしておく必要がある。そのあと、ソースコードをビルドする。

### Clang-3.6 のインストール

Clang-3.6のビルドには、同バージョンのLLVMコアとcompiler-rtが必要である。いずれもhttp://llvm.org/releases/download.html#3.6.2 からバージョン3.6.2が取得できる。また同じページの先頭のリンクでsubversionを用いたリポジトリも公開されている。しかしそれらを利用して取得するよりも、github のミラーを取得したほうが利便性が高い（git checkoutだけで各バージョンのブランチに切り替えることができる）ため、そちらを利用する方法でビルド手順を示す。

% cd 作業用ディレクトリ  
% git clone https://github.com/llvm-mirror/llvm  
% cd llvm  
% git checkout release\_36

% cd 作業用ディレクトリ/llvm/projects  
% git clone https://github.com/llvm-mirror/compiler-rt  
% cd compiler-rt  
% git checkout release\_36

% cd 作業用ディレクトリ/llvm/tools  
% git clone https://github.com/llvm-mirror/clang  
% cd clang  
% git checkout release\_36

% cd 作業用ディレクトリ  
% mkdir build  
 ※llvmのビルド用のディレクトリを準備する。名前は何でもよい。  
% cmake –G Ninja ../llvm  
 ビルド用のサブディレクトリ・ファイル群が生成される。  
　　また、この時点ですでにCコンパイラとしてclangを用いることができる  
　　のであればCC=clang CXX=clang++ という環境変数を設定してcmakeを  
　　おこなってもよい（gcc/g++を用いるよりも高速にビルドが完了する）。  
% ninja  
 ※（PATH環境変数で探せる位置に無いなら適宜フルパスで指定する）  
 ビルドが実行される。マルチコアの場合には自動的に並列ビルドとなる。  
 初回は4コア2スレッドのCPU（8並列）でも二時間ぐらいかかる。  
 明示的に並列度を指定する場合は-jオプションを用いる。ビルドの終盤  
　　でリンクをおこなう際にはメモリ不足になりやすいので、ビルドに失敗  
　　した場合には並列度を下げてやり直すと良い。また、前述のようにして  
　　clang/clang++を用いた方がメモリ消費も少なくて済むようである。  
　　一度ビルドが成功したあとは、再ビルドする際にcmakeから実行する  
　　必要はなく、ninjaコマンドのみを実行すればよい。

以上は通常のRelease版のビルド手順である。これ以外に、デバッグ版のビルドを下記のようにしておこなうことができる。このようにすると、LLVMの各種のコマンド(optなど)で –debug や –debug-only オプションが使えるようになり、コンパイラ内部のデバッグに役立つ情報が得られるようになる。

% cmake –G ninja –DCMAKE\_BUILD\_TYPE=Debug ../llvm  
% ninja  
 ※ビルドを実行（既に実行済みであれば、必要な部分だけリビルド）する。  
 Releaseビルドに比べてさらに多くのメモリを必要とするので、  
 メモリ不足で失敗するようなら-jオプションで並列度を下げて調整する。  
% ninja install  
 ※/usr/local以下にインストールされる。

CMakeのこの他のオプション指定については<http://llvm.org/docs/CMake.html> に書かれているので、そちらを参照されたい。

### CtoXcodeMLのビルド

gitレポジトリから取得した src/ ディレクトリ内で make を実行すればビルドが実行され、 CtoXcodeMLという名の実行ファイルが生成される。

make distcleanをすると、ビルド結果が全て消される。

Make cleanをすると、ビルド結果のうち、もっともビルドに時間のかかるXcodeMlRAV.oを残して、その他を消す。

他にもMakefile内にいくつかターゲットが記述されているが、開発途上段階でのテスト用に準備されたものが多く、動作は保障されない。

## ビルドされたツールの利用方法

例として、

clang -I some-include-dir/ -D user\_defined\_macro=value foo.c

としてコンパイルするような foo.c を対象にする場合、

./CtoXcodeML foo.c -- -I some-include-dir/ -D user\_defined\_macro=value

のように「-- の前に対象ファイル名、 -- のあとにその他のオプション」として指定する。その他のオプションが不要なときも「--」が必要である（この -- が無いと ./compile\_commands.json というファイルが用いられる。これについての詳細は<http://clang.llvm.org/docs/JSONCompilationDatabase.html>を参照のこと）。

以下は、 -- よりも左側に指定することのできるCtoXcodeML独自のオプションについて解説する。

./CtoXcodeML -help

でオプションの一覧が出る。

-file - emit 'file'

-lineno - emit 'lineno'

-column - emit 'column'

…これらはそれぞれ、出力されるXcodeMLの要素にソースコードのファイル名情報・行番号情報・桁位置情報を付加するオプションである。 C\_Frontと同一の結果を得たい場合には –file と –lineno を指定する必要がある。

-disable-typeTable - disable <typeTable>

-disable-symbols - disable <globalSymbols>, <symbols>

-disable-declarations - disable <globalDeclarations>, <declarations>

…これらはそれぞれ、 XcodeMLの一部の出力を抑制するためのオプションである。主にデバッグ用に用いる。

-trace-typeTable - emit traces on <typeTable>

-fulltrace-typeTable - emit full-traces on <typeTable>

-trace-symbols - emit traces on <globalSymbols>, <symbols>

-trace-declarations - emit traces on <globalDeclarations>, <declarations>

-trace-rav - trace Recursive AST Visitor

…これらはそれぞれ、 XcodeMLの出力中にコメントとしてデバッグ情報を含めるためのオプションである。

-typenamemap=<string> - a map file of typename substitution

…これはXcodeMLのtypeTableで扱われる型名を置換するためのオプションである。置換したい型名と置換後の型名をそれぞれ空白区切りで交互に並べたファイルを準備し、そのファイル名を上記の<string>のところに指定する。ただしファイルの末尾に空白や改行があると正しく読み込めなくなるので、ファイルの最終行は改行無しで終わらなければならない。

これ以外にもいくつかのオプションがあるが動作保障はされない。

# 作業報告

## 納品ソフトウェアの内部実装に関する解説

### 各ソースコードについて

* XcodeMlVisitorBase.cpp, XcodeMlVisitorBase.h  
  下記の各種の \*Visitor.cpp, \*Visitor.h の実装の基底クラスである  
  class XcodeMlVisitorBase を準備している部分。  
  意味的にはこのさらに上位に CRTPパターンで書かれたRecursiveASTvisitor クラスを基底に持つのだが、RecursiveASTvisitorクラスは大量のメソッドを持つため、本当に各種の\*Visitorの基底クラスとして実装するとコンパイル時間が何倍にもなるので、pimpl イディオム相当の class RAVBidirBridge をつかって、下記のXcodeMlRAV のほうに RecursiveASTvisitor の実装の部分を隠蔽している。（CRTPパターンとpimplイディオムについては後述）
* XcodeMlRAV.cpp, XcodeMlRAV.h  
  clang の libtooling ライブラリ内の RecursiveASTvisitor.h を利用したクラスを実装している部分。 RAVBidirBridge をつかってclass XcodeMlVisitorBase との間で双方向に橋渡しをしている。
* TypeTableVisitor.cpp, TypeTableVisitor.h  
  clang の AST から XcodeML の <typeTable> 部を生成する部分。  
  また、 class TypeTableInfo というデータ構造を作成し、clang AST の QualType で示された値 (型の種別情報) とXcodeML の の type 名 (文字列) との対応関係を管理する。
* SymbolTableVisitor.cpp, SymbolTableVisitor.h  
  clang の AST から XcodeML の <globalSymbols> 部および <symbols> 部を生成する部分。
* DeclarationsVisitor.cpp, DeclarationsVisitor.h  
  clang の AST から XcodeML の <globalDeclarations> 部および <declarations> 部を生成する部分。この部分がCtoXcodeMLの中でもっとも大きな部分を占める。
* CtoXcodeML.cpp  
  main 関数部分。与えられたコマンドラインから AST を構成し、上記各 Visitor にAST を渡す部分。

### 実装に用いているC++デザインパターンについて

これらのソースコードでは、C++のデザインパターンとして下記の二つのテクニックを用いている。

#### CRTP (Curiously Recurring Template Pattern)

親クラスとして  
　　template <class T> class 親クラス { … };  
のようなものを準備しておいて、利用者は  
　　class 利用者クラス : public 親クラス<利用者クラス> { … };  
のようにして用いる、というデザインパターン。

「奇妙に再帰的なテンプレート」パターンという名前のとおり、親クラスの型引数のところに子クラスの型をいれてそれを継承するという、一見するととても奇妙な形をしている。これは、 template を使わずに下記のように書いた場合と、やりたいことはだいたい同じだと考えてよい。

まず親クラスを  
　　class 親クラス {  
 virtual 仮想メソッド(…) {デフォルトの定義};  
 普通のメソッド(…) { 上記仮想メソッドを使って何か実装; };  
 };  
と定義しておいて、このクラスを継承して利用する側では  
　　class 利用者クラス : public 親クラス {  
 仮想メソッド(…) override { 何か自分用の中身を実装; };  
 };  
のようにする。これにより、親の「普通のメソッド」の動作の一部が利用者クラス側で変更可能となる。

これに対し、CRTPを用いた書き方は次のようになる。

まず親クラスを  
　　template <class D> class 親クラス {  
 置き変え可能なメソッド(…) {デフォルトの定義};  
 普通のメソッド(…) {  
 static\_cast<D\*>(this)->置き変え可能なメソッド(…);  
 という形で呼び出して実装;  
 };  
 };  
としておいて、利用者側で  
 class 利用者クラス : public 親クラス<利用者クラス> {  
 置き変え可能なメソッド(…) { なんか自分用の中身を実装; };  
 };  
とする。これにより、  
 利用者クラス x;  
 x.普通のメソッド(…);  
という形で「普通のメソッド」を呼び出すと、テンプレート展開の結果  
 static\_cast<利用者クラス\*>(this)->置き変え可能なメソッド(…);  
が中で使われることになるので、結果的に「利用者クラス」の側で定義した「置き変え可能なメソッド」が用いられることになる。

virtual を使った場合との違いは下記のようになる。

* virtual を使うと、仮想関数テーブルという「関数のポインタ一覧」がオプジェクトの一部となって埋め込まれているので、 virtual メソッドの呼び出しはメモリアクセスを何度かやって関数ポインタを取得する、というオーバーヘッドがかかる。  
  一方、 CRTP では「テンプレートの実体化」の時点では型引数が静的に定まっているので、メソッド呼び出しは C の関数呼び出しと同等の「静的呼び出し」になる (静的呼び出しはインライン展開できるので、中身が小さいメソッドでは virtual の場合に比べて飛躍的に高速)。
* virtual を使っている場合には  
   親クラス \*pointer = new 利用者クラス(…);  
  のような形で基底クラスのポインタによる抽象化が可能だが、CRTP の「親クラス」というのは実際には単一の基底クラスではない (型引数を入れるごとに違う型ができる) ので、このようなことはできない。つまり、実際には「親クラス」というよりは、単に「同じソースコード片となる部分を共通化して必要に応じて展開いるだけ」といった方が良い。

CRTP を一言でいうと「静的な情報を増やしてコンパイル時点でコンパイラが頑張ることで、利用者の自由度と実行時の高速さの両立をはかったもの」と言える。

#### Pimplイディオム (Pointer to implementation)

CRTP とはまったく逆に、コンパイル時点での静的情報をなるべく分離してソースコード変更の際のリコンパイルの量を減らそうというイディオム。

具体的には、ヘッダファイルに  
　　class 外に見せるクラス {  
 private:  
 class 内部的なクラス \*pimpl;  
 public:  
 コンストラクタ();  
 何らかのメソッド(…);

};  
のような形だけを公開し、 \*.cpp の方 で  
 class 内部的なクラス {  
 …実装すべて…;  
 };  
 外に見せるクラス:コンストラクタ() {  
 pimpl = new 内部的なクラス();  
 }  
 外に見せるクラス:何らかのメソッド(…) {  
 pimpl->何らかのメソッド(…);  
 }  
のようにして外のクラス経由で中の実装にアクセスできるようにする。

デザインパターンの Bridge パターンの応用の一種ともいえる。

## テスト用ソースコードによる評価

テスト用ソースコードとして、 <http://www.omni-compiler.org/benchmark.html> にある下記の三つのコードを対象として評価をおこなった。

* High Performance Linpack
* RandomAccess
* STREAM

ただし、いずれも OMP記法やXMP 記法が含まれており、そのままだとCtoXcodeML には扱えないので、該当部分はコメントアウトして用いるものとする。さらに、 #include によるヘッダファイルを全て取り込むと出力結果が肥大化して評価が難しくなるため、構文エラーの回避に必要となる最小限の宣言だけを残して他は全て削除する形で対応することにした。

いずれも、下記の手順で評価をおこなう。

* 上記のようにしてソースコードに加工をおこない、プリプロセス後の結果を入力ファイルとする。
* C\_Front についてはコマンドラインオプション無し、 CtoXcodeMLについては --lineno --file をつけて確認する。-typenamemap= による型名変換は可能な範囲で調整する。
* 下記コマンドで XcodeML を逆変換し、Cソースコードを復元する。  
  /usr/bin/java -Xmx200m -Xms200m -cp \  
   /usr/local/share/xcalablemp/om-common.jar:/usr/local/share/xcalablemp/om-f-back.jar:\  
   /usr/local/share/xcalablemp/om-c-back.jar:/usr/local/share/xcalablemp/om-exc-tools.jar \  
   exc.util.omompx -decomp -xc –gnu
* 結果を目視確認する。

### High Performance Linpackによる評価

HPL-20141118.tar.bz2を取得し展開すると、下記の二種類のＣソースコードが含まれている。

* main.c
* BLAS/nblas.c

このうち、main.c については XMP 記法が多用されており、 CtoXcodeMLが扱える程度にソースコードを加工した結果を C\_Front・CtoXcodeMLの双方でXcodeMLに変換したところ、これをCソースコードに復元する段階で  
　fatal: cannot convert Xcode to XcodeML.  
　Xcode = (XMP\_DESC\_OF:P13 (VAR:int t))  
　java.lang.Exception: Stack trace  
　（以下スタックトレースが出力される）  
のようなエラーが出てしまったので、評価を中止した。

BLAS/nblas.cについては、 #include <stdio.h>と#pragma ompを削除する形で次評価をおこなった（この時点でプリプロセッサディレクティブもコメントもなくなるのでプリプロセッサを通すことなくC\_Frontに通すことができる形となる）。

入力に用いたソースコード：

void nblas\_dswap(const int n, double \*dx, const int incx, double \*dy, const int incy){

int i;

double dtemp;

for(i=0;i<n;i++){

dtemp = dx[i\*incx];

dx[i\*incx] = dy[i\*incy];

dy[i\*incy] = dtemp;

}

}

void nblas\_dcopy(const int n, const double \*dx, const int incx, double \*dy, const int incy){

int i;

for(i=0;i<n;i++){

dy[i\*incy] = dx[i\*incx];

}

}

void nblas\_dscal(const int n, const double da, double \*dx, const int incx){

int i, nincx;

if(n<=0 || incx<=0 ) return;

nincx = n\*incx;

for(i=0;i<nincx;i+=incx)

dx[i] \*= da;

}

C\_Front・CtoXcodeMLを用いて変換後、それぞれCソースに逆変換した結果：  
（左がC\_Frontを用いた場合、右がCtoXcodeMLを用いた場合）

|  |  |
| --- | --- |
| /\*  \* Original Source : nblas-hacked.c  \* Language : C  \* Compiled Time : 2015-09-18 17:31:14  \* Compiler Info : XcodeML/C-FrontEnd  \* Compiler Version : 0.9.1  \*/  # 1 "nblas-hacked.c"  # 1 "nblas-hacked.c"  void nblas\_dswap(int const n, double \* dx, int const incx, double \* dy, int const incy)  {  {  # 2 "nblas-hacked.c"  int i;  # 3 "nblas-hacked.c"  double dtemp;  # 5 "nblas-hacked.c"  for(i = (0); i < n; i++) {  # 6 "nblas-hacked.c"  dtemp = (\*(dx + (i \* incx)));  # 7 "nblas-hacked.c"  (\*(dx + (i \* incx))) = (\*(dy + (i \* incy)));  # 8 "nblas-hacked.c"  (\*(dy + (i \* incy))) = dtemp;  }  }  }  # 13 "nblas-hacked.c"  void nblas\_dcopy(int const n, double const \* dx, int const incx, double \* dy, int const incy)  {  {  # 14 "nblas-hacked.c"  int i;  # 16 "nblas-hacked.c"  for(i = (0); i < n; i++) {  # 17 "nblas-hacked.c"  (\*(dy + (i \* incy))) = (\*(dx + (i \* incx)));  }  }  }  # 22 "nblas-hacked.c"  void nblas\_dscal(int const n, double const da, double \* dx, int const incx)  {  {  # 23 "nblas-hacked.c"  int i;  # 23 "nblas-hacked.c"  int nincx;  # 24 "nblas-hacked.c"  if((n <= (0)) || (incx <= (0))) {  # 24 "nblas-hacked.c"  return ;  }  # 25 "nblas-hacked.c"  nincx = (n \* incx);  # 27 "nblas-hacked.c"  for(i = (0); i < nincx; i += incx) {  # 28 "nblas-hacked.c"  (\*(dx + i)) \*= da;  }  }  } | /\*  \* Original Source : nblas-hacked.c  \* Language : C  \* Compiled Time : 2015-09-18 17:32:13  \* Compiler Info :  \* Compiler Version :  \*/  # 1 "nblas-hacked.c"  # 1 "nblas-hacked.c"  void nblas\_dswap(int const n, double \* dx, int const incx, double \* dy, int const incy)  {  {  # 2 "nblas-hacked.c"  int i;  # 3 "nblas-hacked.c"  double dtemp;  # 5 "nblas-hacked.c"  for(i = (0); i < n; i++) {  # 6 "nblas-hacked.c"  dtemp = (dx[i \* incx]);  # 7 "nblas-hacked.c"  (dx[i \* incx]) = (dy[i \* incy]);  # 8 "nblas-hacked.c"  (dy[i \* incy]) = dtemp;  }  }  }  # 13 "nblas-hacked.c"  void nblas\_dcopy(int const n, double const \* dx, int const incx, double \* dy, int const incy)  {  {  # 14 "nblas-hacked.c"  int i;  # 16 "nblas-hacked.c"  for(i = (0); i < n; i++) {  # 17 "nblas-hacked.c"  (dy[i \* incy]) = (dx[i \* incx]);  }  }  }  # 22 "nblas-hacked.c"  void nblas\_dscal(int const n, double const da, double \* dx, int const incx)  {  {  # 23 "nblas-hacked.c"  int i;  # 23 "nblas-hacked.c"  int nincx;  # 24 "nblas-hacked.c"  if((n <= (0)) || (incx <= (0)))  # 24 "nblas-hacked.c"  return ;  # 25 "nblas-hacked.c"  nincx = (n \* incx);  # 27 "nblas-hacked.c"  for(i = (0); i < nincx; i += incx) {  # 28 "nblas-hacked.c"  (dx[i]) \*= da;  }  }  } |

両者の差は下記の点である。

* 配列アクセス（ソースコードでは dx[i] のように書かれている部分）が、C\_Frontの場合には \*(dx+i) として再現されるが、 CtoXcodeMLの場合には dx[i] として再現される。  
  これは、 C\_Frontの場合には  
   <pointerRef type="double">  
   <plusExpr type="P4">  
   <Var type="P4" scope="param">dx</Var>  
   <Var type="int" scope="local">i</Var>  
   </plusExpr>  
   </pointerRef>  
  のような XcodeMLに変換されるのに対し、 CtoXcodeMLの場合には  
   <arrayRef type="double">  
   <arrayAddr type="P4" scope="param">dx</arrayAddr>  
   <Var type="int" scope="local">i</Var>  
   </arrayRef>  
  のような XcodeMLに変換されるからである（これはXcodeMLの現状の仕様ではどちらでも良いことになっているので問題ない）。
* 最後の方に現れるif文の本体が、C\_Frontの場合には { } で囲われているが、CtoXcodeMLでは囲われていない。  
  これは、 C\_Front の場合には if 文の<then>部の中身の<returnStatement>が<compoundStatement>でくるまれているのに対し、 CtoXcodeMLではソースコードの通り <then> 部に直接 <returnStatement> が所属しているからである。これもCの仕様から考えるとどちらでも問題ないと考えられる。
* いずれも、CtoXcodeMLの方がソースコードへの可逆性に優れている。

### RandomAccessによる評価

RandomAccess-20150622.tar.bz2を取得し展開すると、下記のＣソースコード一つが含まれている。

* RandomAccess.c

ここから #include や #pragma をコメントアウトし、下記の宣言のみをヘッダファイルから抽出することで、C\_Front, CtoXcodeMLのいずれでも XcodeMLへの変換ができるソースコードが得られた。

typedef struct ompi\_datatype\_t \*MPI\_Datatype;

typedef struct ompi\_communicator\_t \*MPI\_Comm;

#define OMPI\_PREDEFINED\_GLOBAL(type, global) ((type) ((void \*) &(global)))

extern unsigned long long ompi\_mpi\_long\_long\_int;

#define MPI\_LONG\_LONG\_INT OMPI\_PREDEFINED\_GLOBAL(MPI\_Datatype, ompi\_mpi\_long\_long\_int)

extern MPI\_Comm ompi\_mpi\_comm\_world;

#define MPI\_COMM\_WORLD OMPI\_PREDEFINED\_GLOBAL( MPI\_Comm, ompi\_mpi\_comm\_world)

void \*stderr;

しかし、この結果をCのソースコードに復元しようとすると、（XcodeMLの出力が出た後に）下記のエラーが出て、結果として得られた.cファイルは0バイトであった。  
Exception in thread "main" java.lang.IllegalArgumentException: xcodeml.c.decompile.XcCompStmtObj  
（以下、スタックトレースが出力される）

XcodeMLの差を目視確認する限りでは、前述のBLAS/nblas.cの場合と同様に配列アクセスの差とcompoundStatementによる包み込みの差、および型名の差（これはtypenamemapをきちんと適用していないことの影響が大きい）、ヘッダファイルを全て#includeしなかったことによる標準ライブラリ関数の型の差（CtoXcodeMLではプロトタイプ宣言がなくともいくつかの関数は暗黙の型を持つ）が見受けられるため、差分の詳細な解析は割愛したが、おおむね一致しているように見られる。

### STREAMによる評価

STREAM-20150601.tar.bz2を取得し展開すると、下記の2種類のＣソースコードが含まれている。

* STREAM-static.c
* STREAM-dynamic.c

前者と後者の違いは、前者がグローバル変数で領域を確保しているのに対して後者はmallocを用いてメモリ管理している、というものであり、本質的な差は見受けられなかったため、よりシンプルな前者の方をベースにして #include や #pragma をコメントアウトして対応をおこなうことにした。ただし、 FLT\_MAXが用いられているので #include <float.h> は残し、これをプリプロセスした結果を用いた。

C\_Front, CtoXcodeMLのいずれでも XcodeMLへの変換ができるソースコードが得られた。

入力に用いたソースコード：

# 1 "STREAM-static-hacked.c"

# 1 "<built-in>" 1

# 1 "<built-in>" 3

# 312 "<built-in>" 3

# 1 "<command line>" 1

# 1 "<built-in>" 2

# 1 "STREAM-static-hacked.c" 2

# 1 "/usr/local/bin/../lib/clang/3.6.2/include/float.h" 1 3

# 10 "STREAM-static-hacked.c" 2

double a[357913941], b[357913941], c[357913941];

void checkSTREAMresults(){

int j, k;

double aj=2.0, bj=2.0, cj=0.0, scalar=3.0, asum=0.0, bsum=0.0, csum=0.0, epsilon=1.e-8;

for(k=0; k<10; k++) aj = bj + scalar\*cj;

aj = aj \* (double)357913941 \* xmp\_num\_nodes();

bj = bj \* (double)357913941 \* xmp\_num\_nodes();

cj = cj \* (double)357913941 \* xmp\_num\_nodes();

for(j=0; j<357913941; j++) {

asum += a[j];

bsum += b[j];

csum += c[j];

}

if(xmp\_node\_num() == 1)

if(fabs(aj-asum)/asum > epsilon || fabs(bj-bsum)/bsum > epsilon || fabs(cj-csum)/csum > epsilon)

printf("Failed Validation %f %f %f\n", fabs(aj-asum), fabs(bj-bsum), fabs(cj-csum));

else

printf("Solution Validates\n");

}

void HPCC\_Stream(double \*triadGBs){

int j, k;

double scalar = 3.0, times[10], mintime = 3.40282347e+38F, curGBs;

for(j=0; j<357913941; j++) {

a[j] = 1.0;

b[j] = 2.0;

c[j] = 0.0;

}

for(j=0; j<357913941; j++)

a[j] = 2.0 \* a[j];

for(k=0; k<10; k++) {

times[k] = -xmp\_wtime();

for (j=0; j<357913941; j++)

a[j] = b[j] + scalar\*c[j];

times[k] += xmp\_wtime();

}

for (k=1; k<10; k++)

mintime = (((mintime) < (times[k])) ? (mintime) : (times[k]));

curGBs = (mintime > 0.0 ? 1.0 / mintime : -1.0);

curGBs \*= 1e-9 \* 3 \* sizeof(double) \* 357913941;

\*triadGBs = curGBs;

checkSTREAMresults();

}

int main(int argc, char \*\*argv){

double triadGBs;

HPCC\_Stream(&triadGBs);

if(xmp\_node\_num() == 1)

printf("[Vector size is %d] Total Triad %.2f GB/s on %d process\n", 357913941, triadGBs, xmp\_num\_nodes());

return 0;

}

C\_Front・CtoXcodeMLを用いて変換後、それぞれCソースに逆変換した結果：  
（左がC\_Frontを用いた場合、右がCtoXcodeMLを用いた場合）

|  |  |
| --- | --- |
| /\*  \* Original Source : STREAM-static-hacked.c  \* Language : C  \* Compiled Time : 2015-09-19 16:27:57  \* Compiler Info : XcodeML/C-FrontEnd  \* Compiler Version : 0.9.1  \*/  # 1 "STREAM-static-hacked.c"  # 13 "STREAM-static-hacked.c"  double a[357913941];  # 13 "STREAM-static-hacked.c"  double b[357913941];  # 13 "STREAM-static-hacked.c"  double c[357913941];  # 15 "STREAM-static-hacked.c"  void checkSTREAMresults()  {  {  # 16 "STREAM-static-hacked.c"  int j;  # 16 "STREAM-static-hacked.c"  int k;  # 17 "STREAM-static-hacked.c"  double aj = 2.0;  # 17 "STREAM-static-hacked.c"  double bj = 2.0;  # 17 "STREAM-static-hacked.c"  double cj = 0.0;  # 17 "STREAM-static-hacked.c"  double scalar = 3.0;  # 17 "STREAM-static-hacked.c"  double asum = 0.0;  # 17 "STREAM-static-hacked.c"  double bsum = 0.0;  # 17 "STREAM-static-hacked.c"  double csum = 0.0;  # 17 "STREAM-static-hacked.c"  double epsilon = 1.e-8;  # 19 "STREAM-static-hacked.c"  for(k = (0); k < (10); k++) {  # 19 "STREAM-static-hacked.c"  aj = (bj + (scalar \* cj));  }  # 21 "STREAM-static-hacked.c"  aj = ((aj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 22 "STREAM-static-hacked.c"  bj = ((bj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 23 "STREAM-static-hacked.c"  cj = ((cj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 25 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 26 "STREAM-static-hacked.c"  asum += (a[j]);  # 27 "STREAM-static-hacked.c"  bsum += (b[j]);  # 28 "STREAM-static-hacked.c"  csum += (c[j]);  }  # 32 "STREAM-static-hacked.c"  if((xmp\_node\_num()) == (1)) {  # 33 "STREAM-static-hacked.c"  if(((((fabs(aj - asum)) / asum) > epsilon) || (((fabs(bj - bsum)) / bsum) > epsilon)) || (((fabs(cj - csum)) / csum) > epsilon)) {  # 34 "STREAM-static-hacked.c"  printf("Failed Validation %f %f %f\n", fabs(aj - asum), fabs(bj - bsum), fabs(cj - csum));  }  else {  # 36 "STREAM-static-hacked.c"  printf("Solution Validates\n");  }  }  }  }  # 39 "STREAM-static-hacked.c"  void HPCC\_Stream(double \* triadGBs)  {  {  # 40 "STREAM-static-hacked.c"  int j;  # 40 "STREAM-static-hacked.c"  int k;  # 41 "STREAM-static-hacked.c"  double scalar = 3.0;  # 41 "STREAM-static-hacked.c"  double times[10];  # 41 "STREAM-static-hacked.c"  double mintime = (float)3.40282347e+38F;  # 41 "STREAM-static-hacked.c"  double curGBs;  # 43 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 44 "STREAM-static-hacked.c"  (a[j]) = (1.0);  # 45 "STREAM-static-hacked.c"  (b[j]) = (2.0);  # 46 "STREAM-static-hacked.c"  (c[j]) = (0.0);  }  # 49 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 50 "STREAM-static-hacked.c"  (a[j]) = ((2.0) \* (a[j]));  }  # 52 "STREAM-static-hacked.c"  for(k = (0); k < (10); k++) {  # 53 "STREAM-static-hacked.c"  (times[k]) = (-(xmp\_wtime()));  # 59 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 60 "STREAM-static-hacked.c"  (a[j]) = ((b[j]) + (scalar \* (c[j])));  }  # 62 "STREAM-static-hacked.c"  (times[k]) += (xmp\_wtime());  }  # 65 "STREAM-static-hacked.c"  for(k = (1); k < (10); k++) {  # 66 "STREAM-static-hacked.c"  mintime = ((mintime < (times[k])) ? mintime : (times[k]));  }  # 68 "STREAM-static-hacked.c"  curGBs = ((mintime > (0.0)) ?((1.0) / mintime) : (-(1.0)));  # 69 "STREAM-static-hacked.c"  curGBs \*= ((((1e-9) \* (3)) \* (sizeof(double))) \* (357913941));  # 70 "STREAM-static-hacked.c"  (\*(triadGBs)) = curGBs;  # 72 "STREAM-static-hacked.c"  checkSTREAMresults();  }  }  # 75 "STREAM-static-hacked.c"  int main(int argc, char \* \* argv)  {  {  # 76 "STREAM-static-hacked.c"  double triadGBs;  # 78 "STREAM-static-hacked.c"  HPCC\_Stream(&(triadGBs));  # 81 "STREAM-static-hacked.c"  if((xmp\_node\_num()) == (1)) {  # 82 "STREAM-static-hacked.c"  printf("[Vector size is %d] Total Triad %.2f GB/s on %d process\n", 357913941, triadGBs, xmp\_num\_nodes());  }  # 84 "STREAM-static-hacked.c"  return 0;  }  } | /\*  \* Original Source : STREAM-static-hacked-pp.c  \* Language : C  \* Compiled Time : 2015-09-25 11:57:06  \* Compiler Info :  \* Compiler Version :  \*/  # 1 "STREAM-static-hacked-pp.c"  # 13 "STREAM-static-hacked.c"  double a[357913941];  # 13 "STREAM-static-hacked.c"  double b[357913941];  # 13 "STREAM-static-hacked.c"  double c[357913941];  # 15 "STREAM-static-hacked.c"  void checkSTREAMresults()  {  {  # 16 "STREAM-static-hacked.c"  int j;  # 16 "STREAM-static-hacked.c"  int k;  # 17 "STREAM-static-hacked.c"  double aj = 2.0;  # 17 "STREAM-static-hacked.c"  double bj = 2.0;  # 17 "STREAM-static-hacked.c"  double cj = 0.0;  # 17 "STREAM-static-hacked.c"  double scalar = 3.0;  # 17 "STREAM-static-hacked.c"  double asum = 0.0;  # 17 "STREAM-static-hacked.c"  double bsum = 0.0;  # 17 "STREAM-static-hacked.c"  double csum = 0.0;  # 17 "STREAM-static-hacked.c"  double epsilon = 1.e-8;  # 19 "STREAM-static-hacked.c"  for(k = (0); k < (10); k++) {  # 19 "STREAM-static-hacked.c"  aj = (bj + (scalar \* cj));  }  # 21 "STREAM-static-hacked.c"  aj = ((aj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 22 "STREAM-static-hacked.c"  bj = ((bj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 23 "STREAM-static-hacked.c"  cj = ((cj \* ((double)(357913941))) \* (xmp\_num\_nodes()));  # 25 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 26 "STREAM-static-hacked.c"  asum += (a[j]);  # 27 "STREAM-static-hacked.c"  bsum += (b[j]);  # 28 "STREAM-static-hacked.c"  csum += (c[j]);  }  # 32 "STREAM-static-hacked.c"  if((xmp\_node\_num()) == (1))  # 33 "STREAM-static-hacked.c"  if(((((fabs(aj - asum)) / asum) > epsilon) || (((fabs(bj - bsum)) / bsum) > epsilon)) || (((fabs(cj - csum)) / csum) > epsilon))  # 34 "STREAM-static-hacked.c"  printf("Failed Validation %f %f %f\n", fabs(aj - asum), fabs(bj - bsum), fabs(cj - csum));  else  # 36 "STREAM-static-hacked.c"  printf("Solution Validates\n");  }  }  # 39 "STREAM-static-hacked.c"  void HPCC\_Stream(double \* triadGBs)  {  {  # 40 "STREAM-static-hacked.c"  int j;  # 40 "STREAM-static-hacked.c"  int k;  # 41 "STREAM-static-hacked.c"  double scalar = 3.0;  # 41 "STREAM-static-hacked.c"  double times[10];  # 41 "STREAM-static-hacked.c"  double mintime = (float)3.40282347e+38F;  # 41 "STREAM-static-hacked.c"  double curGBs;  # 43 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 44 "STREAM-static-hacked.c"  (a[j]) = (1.0);  # 45 "STREAM-static-hacked.c"  (b[j]) = (2.0);  # 46 "STREAM-static-hacked.c"  (c[j]) = (0.0);  }  # 49 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 50 "STREAM-static-hacked.c"  (a[j]) = ((2.0) \* (a[j]));  }  # 52 "STREAM-static-hacked.c"  for(k = (0); k < (10); k++) {  # 53 "STREAM-static-hacked.c"  (times[k]) = (-(xmp\_wtime()));  # 59 "STREAM-static-hacked.c"  for(j = (0); j < (357913941); j++) {  # 60 "STREAM-static-hacked.c"  (a[j]) = ((b[j]) + (scalar \* (c[j])));  }  # 62 "STREAM-static-hacked.c"  (times[k]) += (xmp\_wtime());  }  # 65 "STREAM-static-hacked.c"  for(k = (1); k < (10); k++) {  # 66 "STREAM-static-hacked.c"  mintime = ((mintime < (times[k])) ? mintime : (times[k]));  }  # 68 "STREAM-static-hacked.c"  curGBs = ((mintime > (0.0)) ?((1.0) / mintime) : (-(1.0)));  # 69 "STREAM-static-hacked.c"  curGBs \*= ((((1e-9) \* (3)) \* (sizeof(double))) \* (357913941));  # 70 "STREAM-static-hacked.c"  (\*(triadGBs)) = curGBs;  # 72 "STREAM-static-hacked.c"  checkSTREAMresults();  }  }  # 75 "STREAM-static-hacked.c"  int main(int argc, char \* \* argv)  {  {  # 76 "STREAM-static-hacked.c"  double triadGBs;  # 78 "STREAM-static-hacked.c"  HPCC\_Stream(&(triadGBs));  # 81 "STREAM-static-hacked.c"  if((xmp\_node\_num()) == (1))  # 82 "STREAM-static-hacked.c"  printf("[Vector size is %d] Total Triad %.2f GB/s on %d process\n", 357913941, triadGBs, xmp\_num\_nodes());  # 84 "STREAM-static-hacked.c"  return 0;  }  } |

上記の結果も、 nblasと同様、配列アクセスの差と compoundStatementの包み込みの差であり、いずれも本質的な問題ではないと考えられる。