最適化の正しさを検証する

先進のコンパイラ最適化が安定していて十分テストされているものだと考えることは間違っています。最近、最適化のテストを行う中で私どもが入手できるどのコンパイラ技術においてもエラーが存在することを確認しました。このことから、現状では先進の最適化テストがコンパイラ開発者にとって発展途上の技術であり、対処する必要があるのだと結論づけられます。

本資料では、最先端のコンパイラテストスイート SuperTest における新しい最適化テストを紹介します。

コンパイラの最適化には非常に高い経済的価値があります。最適化されていないものと比較すると、最適化されたコードによるプログラムは 15 倍の実行速度の向上を示すことがあります。大きな倍数ですが、ベクトル化のような高度なループ最適化であればこれは往々にして得られるものです。経済的価値の点では 15 倍の実行効率向上は、より低速でずっと安価なプロセッサの使用や熱放出の減少、コンパクトな構造、そして同時に（バッテリー）電力消費を 15 倍も削減できることを意味するものです。このことが組込みや車載アプリケーションで実現すれば、コンパイラ最適化による真の利益が得られます。

通常コンパイラはそのソースコードの半分以上が最適化に関連するものです。これは重要なことであり、コンパイラのエラーが他の部分と同様に、この最適化処理の部分で発生するということです。

1 「最適化の言語仕様は存在しない」

この 1 年私どもは SuperTest の最適化テスト能力の向上を図ってきました。SuperTest のテストも含めテストを作るときは言語仕様から始めることが望ましいことですが、最適化に対してこれは簡単ではありません。C/C++の言語仕様の観点では最適化はほとんど存在しないのです。C11 の 5.1.2.3 には、こうあります：

> *この規格における意味規則の規定では，最適化の問題を無視して抽象計算機の動作として記述する。*[1]

言語規格は言語要素個々の動作を規定していますが、その動作がいつ、どのようにして達成されるかは規定していません。最適化はいわゆる「非機能」要求なのです。このことで、仕様に対する最適化を検証することが不可能ではないにせよ困難になっています。

多くのコンパイラのテストで使用されてきた経験から、SuperTest が最適化のテストで常に非常によい仕事をしていると確信しています。SuperTest の多くのテストはコンパイラ最適化のテストのために設計されています。例えば、SuperTest のスイートで「tail recursion」とテキストサーチするとすぐに 10 個ほどのテストを見つけられます。この中にはたまたま末尾再帰をテストするものや、末尾再帰をテストしていると文書化されなかったものは含まれていません。コンパイラの処理はパイプラインになっており、各テ

[1] (訳注) 原文が同文である C99 の JIS 規格（X3010:2003）の 5.1.2.3 を掲載しています。

ストは最適化のステージも含めてパイプラインのすべてのコンポーネントに触れることになります。これはテストが意図せずに最適化に遭遇する確率が高いことを意味し、実際その通りになっていることを確認しています。

このことの弱点は、もっともなことですが機能安全規格の形式的な要求には十分ではないことです。それらの標準は「偶然」の少ないアプローチを要求しており、テストと最適化の要件をリンクする厳密なフレームワークが必要になります。このフレームワークを提供することこそ私どもが始めたことで、その目標は単に機能安全規格の要件に合致することだけでなく、広く適用できる高品質な最適化テストスイートを実際に作成することです。

2　ベンチマークからの着想、教訓

最適化が適用されるコードを思い浮かべるためにベンチマークを考えます。いくつかはパフォーマンステストの分野でよく知られているものです。明らかに最適化ができるように記述された（わざとらしい）ベンチマークばかりあるのではなく、コンパイラ開発者は最適化を追加することでベンチマークコードの性能が向上するよう励んでいます。このことから以下のいくつかの教訓が得られ、それらをまとめると次のようになります：

ベンチマークはコンパイラ最適化の正しさの最良のテストではない

教訓 1：　ベンチマークは常に結果を検証するものとは限らない

ベンチマークの第一の基準はコンパイルされたコードがターゲットプロセッサでどれほど速く実行できるかで、第二はありません。計算されるものが正しいかどうかを検証することに常に高い優先度が与えられるとは限りません。ではどうしたらベンチマークになされた最適化が正しく動作していると分かるでしょうか？グラフィックスのベンチマークでは生成されたイメージを見て正しさを評価できるものもあるでしょうが、そのようなアプローチが継続的インテグレーション環境でうまく働かないのは明らかです。

教訓 2：　ベンチマークは異なるデータモデルを扱うようには書かれていない

SuperTest の開発において、組込みの基本システムでは多くの異なるデータモデルが使用されているといつも認識していました。（32 ビット整数の代わりに 24 ビットを用いるなど）異なるデータモデルで計算を行うことは、異なった、正しくないかもしれない結果をもたらすことがあります。最適化においてはこのことを考慮しなければなりません。しかし教訓 1 を振り返りますと、計算結果が検証されないと最適化がデータモデルを考慮している証拠を持てないのです。

教訓 3：　ベンチマークは未定義の動作から自由にはなれない

ベンチマークが想定しているよりも小さいデータモデルでコンパイルされたとき、未定義の動作が容易に発生します。このことの危険性は、コンパイラの中には最適化を受ける特性に未定義の動作がないものとして扱うものがあるということです。例えば、コード内の分岐がコンパイラによって未定義の動作（符号付整数演算のオーバーフローで引き起こされるなど）につながると解析された場合、コンパイラはもっと

もなこととして、その分岐が決して実行されないものであると想定して削除するかもしれません。これはベンチマークが意図したことではありませんが、実行は速くなります。

教訓 4： ベンチマークは生成されたコードすべてを実行するのではない

たとえベンチマークで計算結果が正しいと検証されても、生成コードのすべてが実行されていないのであれば、実行されていないコードを生成した変換が正しく行われたという証拠はありません。これは最適化で削除されるベンチマーク内の「不要コード」の単純な事例ではないのです。反対に、最適化なしにコンパイルされた場合には生成されたコードはすべて実行されるでしょう。最適化された後に実行されないコードが存在する訳は、最適化変換ではコードを複製したり特殊化したりすることが多いからです。この過程でずっと複雑な制御フローが構築され、特殊化の事例を区別して高速にします。ベンチマークには特殊化事例のすべてがあるわけではないとすれば、生成コードの一部は実行されないことになります。

以上から、コンパイラがベンチマークスイートを本当に上手く最適化するとしても、その最適化が正しいという保証はないということが明らかでしょう。

これらの「教訓」が SuperTest の最適化テストスイートを作るときに活かされています。私どもはできうる限り計算結果が検証されること、サポートしているデータモデルに対してテストが適用できること、テストに未定義の動作を含まないこと、そしてテストがすべての生成コードを実行すること、を確かめてきました。

3　生成されたコードすべてを実行すること（教訓 4）

すべてのベンチマークにこれらの欠陥があるわけではありません。例えば EEMBC の CoreMark® では計算結果を検証するのに苦労を惜しんでいません。しかし、すべての生成コードに対する実行が不完全であることが本当に問題であって、私どもが知るどんなベンチマークでも扱われていないものなのです。

テスト結果を検証すること、正しいデータモデルを使うこと、そして未定義動作がないこと、はテストのソースコードがもつ特質すべてです。しかし、最適化後に生成コードのすべてが実行されることはコンパイラが適用する変換に依存します。我々は前もってそれを知ることはできせんので、生成されるコードのすべてが実行されることを保証できませんが、そうできるように懸命に取り組んできました。

生成コードの実行を解析するため、異なるコンパイラいくつかを使ってアセンブリ言語レベルで実行時のカバレッジをとりました。実際には、構造カバレッジとブランチカバレッジの両方を調べて、アセンブリ言語レベルの MC/DC 分析を実行しました。次に、可能なすべての最適化について、コンパイラが適用するコード特殊化を調べ、生成された特殊化コードすべてに的中するのに必要な個別の入力を解析しました。

例えば、次の比較的簡単なループを考えます：

```
int f(int n)
{
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += i & n;
```

```
    }
}
```

このループは十分に複雑ですのでコンパイラはこれを完全に削除する代数的な解析は行いません。その代わりにこのループはベクトル化できます。ソースコードのレベルでこのループを 0 以外の値の n で 1 回呼び出せば、コードの完全な構造カバレッジがとれます。ループ条件に基づく分岐一つは実行時に両方向が呼ばれ、完全なブランチカバレッジが取れます。

高い最適化レベルで生成されたアセンブリコードを見ますとコードはかなり展開されており、条件分岐が 15 個もあることがわかります。このループを n=999（ループ展開数の倍数やベクトル長でない意味のある数値）で呼び出すと約 80%のコードカバレッジ（まあいい値です）が得られますが、生成コードの制御フローの枝の半数未満のカバレッジとなります（これはよくありません）。

最大のコードカバレッジ、ブランチカバレッジを達成するには最適化コードは少なくとも *5* つの異なる繰り返し数値 n で呼ばれないといけません。

興味深く、多分驚くべき発見は、コンパイラが冗長な条件分岐を生成していることです。これは複数の異なるコンパイラ技術において見られたことですので、単に一つのコンパイラの生成物の話ではありません。上記のループでも起こっています。冗長な分岐は制御フローの中で先に作られていた条件を再検査します。その結果、条件が常に同じ値となってこの冗長な条件分岐が常に同じ方向となります。厄介なことに、このような分岐に対する完全なブランチカバレッジは達成できません！　また、コードカバレッジとブランチカバレッジを 100%にできないことに加えて、冗長な分岐は解析が難しく、カバレッジが 100%にならないことを調査するのに多くの時間を費やすことになります。

これらをすべて考慮して、すべてのテストに対して対象としているコンパイラすべてに最大のカバレッジを達成するテストケースの範囲を構築する一般化した方法を作成することができました。その方法は強固で個別のコンパイラ技術に依存せずに動作するので、私どものテストとテストケースはどのようなコンパイラに対しても同様に最大のカバレッジを達成することは間違いありません。

4　結果

これまでのものと新しい最適化のテストを用いて、今まで解析してきたすべてのコンパイラ技術の最適化エラーを見つけています。これは、最適化が一般的に信頼できないものであるということではなく、注意深く検討することが賢明であるということを示唆するものです。

次の例は組込みシステムでよく使用されているあるコンパイラに対する SuperTest のテスト結果です：

```
s[0] = 42;
*(sp[0]) = -1;    /* *(sp[0]) は s[0] の別名 */
if (s[0] == 42) { /* 誤って true となる */
```

このエラーは、第 2 文で別名による変更があるにも関わらず、変数 s[0]を通じた最初の文から 3 番目の文への間違った値伝播の結果です。このコンパイラのエラーには、コンパイラに与える最適化オプションで指定できずにこの最適化が適用されてしまうという危険性があります。最適化が起きていないと思ってい

ても起こっているのです。言語の定義に関する限りこれは問題ありませんが、多くの開発者はこのような事態に遭遇するとは想像もしないでしょう。

次は Intel の ICC（Intel C++ Compiler）でコンパイルされた別な最適化スイートの結果です。このコンパイラは高性能なループ最適化で高い評価を受け、よく知られていますが、ここに示すように完璧ではありません：

```
void s482(double *a, double *b, double *c, int len)
{
    int i;
    for (i = INT_MIN; i < INT_MIN+len; i++) {
        a[i-INT_MIN] = b[i-INT_MIN] + c[i-INT_MIN];
        if (c[i-INT_MIN] > b[i-INT_MIN])
            break;
    }
}
```

このテストに特有の性質は、その反復子 `i` が、最適化機能が想定していない整数の値の範囲の下限に近いところで動作することです。その結果プログラムは実行時にセグメンテーションフォールトでクラッシュします。CLANG/LLVMやGCC、Microsoftのコンパイラといった他の信頼できる技術でも別な最適化エラーが見つかりました。

5　結論

最も重要な結論は、コンパイラ最適化を利用するならコンパイラの弱点を知らなければならないということです。最適化エラーを免れないコンパイラ技術を見たことはありません。目立たず、容易に避けることができるエラーもありますが、特定の最適化を使用しないことを考えなければならないほど深刻なものがあります。性能解析のためのベンチマークスイートだけでなく、コンパイラ最適化の正しさを検証するためのテストスイートも忘れないでください。

もし機能安全やミッションクリティカルな要求に対してコンパイラ最適化の使用を認証することが必要でしたら、私どもの新しいテストスイートが強固な基盤を提供します。最適化はコンパイラの機能要件ではなくコンパイラ技術ごとに固有の最適化実装方針があります。私どもの最適化テストスイートは SuperTest のコアスイートで多くの最適化が適用されるテストと併用して、あらゆるコンパイラで、すべてではないにしても、多くの最適化のテストをカバーします。

FUJI SETSUBI
富士設備工業株式会社 電子機器事業部　www.fuji-setsu.co.jp