組込み向け言語 Continuation based C の GCC 上の実装

Implementation of the Continuation based C on GCC

088511J　与儀健人　　　　指導教官: 河野真治

1　はじめに

企業システムの多様化、IT 導入の加速により、ソフトウェアは大規模化・複雑化する傾向にある。また家電製品のデジタル化も進み、組み込みシステムの需要も増大している。

それにともないハードウェアは驚異的な進歩を遂げてきた。しかしハードウェアの進歩に対し、ソフトウェアはその進歩に追いついていない。オブジェクト指向が発明され、Java などが注目されているが、ガベージコレクタや実行時コンパイルは余分な処理が必要となる。軽量かつ高速な応答が要求される Real-time 処理や組込み用途には適さない。

PlayStation3 には Cell という特殊な CPU が採用され注目されている。しかしプログラミングは格段に難しく複雑になった。

ハードウェアの進化や数学的検証にソフトウェアが対応するためには、これまでとは違う新たな視点を持ったプログラミング言語が望ましい。

我々はこれらの問題に取り組むため、Continuation based C（以下 CbC）という言語を提案している。Continuation とはプログラムの次の実行処理を表現する制御構造で、継続とも呼ばれている。CbC では C からサブルーチンやループ制御を除き、代わりに継続をベースとした実行制御を行う。

これまで CbC のコンパイルには、micro-c をベースとしたコンパイラが用いられてきた。加えて 2008 年の研究において GCC をベースとした CbC コンパイラが開発され、継続処理の実装が行われた。

本研究では GCC ベースのコンパイラにおいて残る CbC の機能の実装を行い、実用的な CbC プログラムの動作を目指す。

2　Continuation based C (CbC)

CbC は、スタックを保持しない継続、“軽量継続” をプログラミング記述のベースとした言語である。関数の代わりとなるそれぞれの処理単位は “コードセグメント” と呼ばれる。CbC ではこのコードセグメントにより、状態遷移を直接プログラムとして記述することができる。

以下では簡単な CbC 記述の例として階乗計算を行うコードセグメントを例示する。

```
code factor0(int prod, int x,
             code (*next)(int)) {
  if (x >= 1) {
    goto factor0(prod*x, x-1, next);
  } else {
    goto (*next)(prod);
  }
}
```

この例の様にコードセグメントへの継続では、自分自身に対して継続することでループ制御を実現する事ができる。また、例にあるようにポインタの参照先に継続する “間接継続” も可能になっている。

3　軽量継続の実装方法

初代の CbC コンパイラである micro-c は元より軽量継続を意識して開発されており、コードセグメントに適した設計がなされている。しかし GCC ではメンテナンス性の理由からそのような深いレベルでの実装は好ましくない。既に入念に設計され実際に使われている関数と関数呼び出しを利用して軽量継続を実装する。

3.1　末尾呼出による軽量継続

末尾呼出とはリターン文直前の関数呼び出しのことで、GCC の最適化の一つである。通常の関数呼び出しは復帰後に元の環境に戻るが、この末尾呼び出しの場合はその必要がなく、call 命令の代わりに jmp 命令を使うことができる。そしてスタックを余計に積むこともない（図 1）。

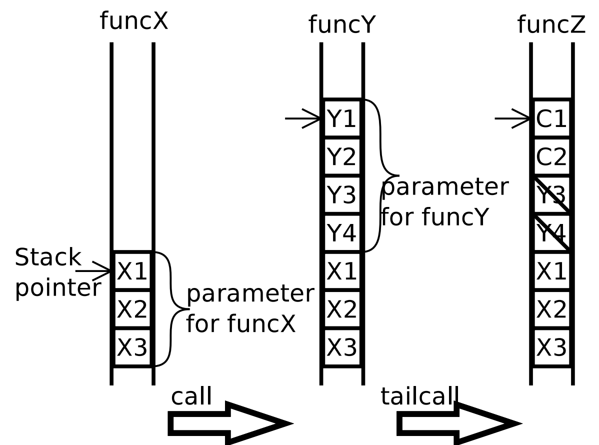


図 1: 末尾呼出と通常呼出のスタックの変化

この特徴は軽量継続のそれとほぼ同じである。構文解析にてこの最適化を強制した関数呼出を生成することで、軽量継続の実装ができる。

3.2　fastcall による高速化

以上で軽量継続は可能になったが、これだけでは CbC 用に入念に設計された micro-c よりも良い性能を出すことはできない。特に x86 アーキテクチャにおいての高速化を行う必要がある。x86 の関数呼出規約では全ての引数はスタックに確保するため、メモリアクセスが多い。 fastcall を用いてこの関数規約を変更しスタックでなくレジスタを使用するように変更する。

これによりメモリアクセスが減り、十分な高速化が得られた。6.2 には測定結果が見られる。

4　環境付き継続の実装

既存のソフトウェアを無駄にしないためにも、新しい言語が受け入れられるためにも、既存の言語との互換性は必須である。 CbC では環境付き継続という形で C との互換性を担保している。

こちらは C の関数内から先ほどのコードセグメント factor0 に継続する例である。

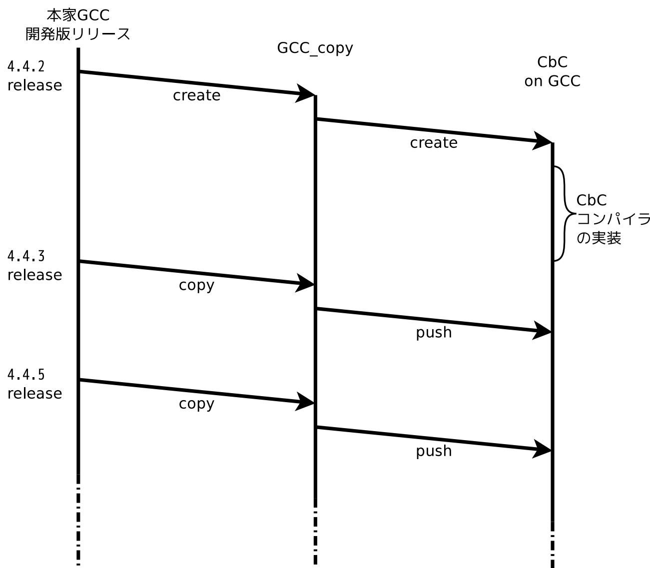
```
int caller() {
  goto factor0(1, i, __return);
}
```

本来は継続した場合、元の環境に復帰することはできないが、ここでは復帰のために `__return` という特殊なコードセグメントを factor0 に渡している。この特殊なコードセグメントは `factor0` が処理を終える際に間接継続として引数 x と共に継続対象となり、その時、関数 `caller` は返り値 x を伴って復帰する。

この環境付き継続の実装には setjmp()/longjmp() を使った方法も考えられるが、ポータビリティのため、ここでは GCC の拡張機能でもある内部関数を用いて実装を行った。

5　メンテナンス性の向上に関する取り組み

新しいコンパイラは GCC をベースとした。GCC の本家でのアップデートリリースは年 5 回ほどあり、CbC コンパイラもこれに追従するのが望ましい。



このメンテナンスのため、CbC コンパイラの管理に分散バージョン管理を用い、GCC 本家のリリースを追従するリポジトリと CbC 開発用のリポジトリの 2 つを管理する手法を用いた。この手法により、アップデートの手順が明確になり、重要な変更点のみに集中できるようになった。

6　評価

6.1　micro-c との速度比較

GCC ベースのコンパイラと micro-c をベースとしたコンパイラで生成した実行ファイルの速度差を比較する。次の表がその結果である。x86 に特化したコンパイラである micro-c と

	GCC		micro-c
	最適化なし	速度最適化	
x86/OS X	5.901	2.434	2.857
x86/Linux	5.732	2.401	2.254
ppc/OS X	14.875	2.146	4.811
ppc/Linux	19.793	3.955	6.454
ppc/PS3	39.176	5.874	11.121

ほぼ伍角の速度を得られた。また PowerPC においてはいずれの環境でも micro-c の倍近い速度を計測することができた。

6.2　前バージョンとのとの速度比較

次に、前回の実装時における GCC ベースコンパイラと、今回改善したコンパイラとの速度比較を次の表に示す。

	新バージョン		前バージョン	
最適化	なし	あり	なし	あり
x86/OS X	5.907	2.434	4.668	3.048
x86/Linux	5.715	2.401	4.525	2.851

新バージョンでは最適化を行わない場合に速度の低下が見られた。これは末尾呼出の強制のために行った処理が影響したものであり、予想通りの結果であった。この速度低下は最適化によりカバーされ得る。実際に最適化を行った場合は 15–20%ほど旧バージョンより高速化に成功している。こちらは fastcall による影響だと考えられる。

7　まとめと今後の課題

本研究による成果を以下にあげる。

- GCC をベースとした CbC コンパイラが CbC のフルセットとして利用可能となった
- CbC が 20 以上の多数のアーキテクチャに対応
- CbC の高速化（特に x86 について大幅に改善された）
- デバッガとして gdb が使用可能になった

今後の課題を以下に述べる。

- Real-time、組込み向けに実用的な例題の作成
- タブロー法を用いた検証手法の確立
- オブジェクティブな CbC の設計
- スケジューラを使った CbC の並列化

参考文献

[1] 河野真治. “Implementing Continuation based language in GCC”, Continuation Festa 2008, April, 2008

[2] 与儀健人, 河野真治. “組み込み向け低レベル言語 CbC の GCC による実装”, 第 6 回ディペンダブルシステムワークショップ, July, 2008

[3] 河野真治. “検証を自身で表現できるハードウェア、ソフトウェア記述言語 Continuation based C と、その Cell への応用”, 電子情報通信学会 VLSI 設計技術研究会, March, 2008