Continuation based C によるPS3 Cellのシミュレーション

神里　晃[†]　河野真治[††]

我々は状態遷移記述に向いたCの下位言語である Continuation based C(CbC) を提案, 実装している.CbC はデバイスドライバや OS の記述, または最適化自体を記述するのに向いているが, ハードウェアの記述自体も可能である. ここでは PS3 に内蔵された Cell Chip の記述を行い, その有効性を示す. これにより,CbC で記述された Cell は, 実機やエミュレータ抜きで性能を評価することが可能となる.

The simulation of PS3 Cell by Continuation based C

Akira Kamizato[†] and Shinji Kono[††]

We are proposing and realizing Continuation based C(CbC),which is a low level language of C. CbC is a suitable for describing state machine such as device driver and operating system and optimization itself. It is also possible to describe hardware. Here we show a description of the Cell chip which is embedded PS3. Now I don't have PS3,but We evalute the efficiency without emulator and substance machine.

1. はじめに

近年家庭用ゲーム機は特殊なハードウェアで構成される. このようなハードウェアは開発時間が重要な問題となる. ハードウェアの開発と同時にソフトウェアの開発が同時に行われることが望ましい. またゲームプログラミングに置いて, 特殊なハードウェアに置けるアセンブラでプログラミングすることは必須となる. ここでは通常のプログラム言語よりも下位に位置する言語, Continuation based C(以下 CbC) を用いて, 環境に依存しないシミュレータを作成し, 開発時間の短縮, アセンブラのデバッグなどに CbC が有効であることを示す.

表現するターゲットとして現在 SCEI で開発中の PlayStation3(以下 PS3) を採用する. PS3 は Linux が搭載される予定で, ユーザによるプログラミングが可能である. PS3 は PowerPC アーキテクチャを中心に 8 つの Synergistic Processor Unit(以下 SPU) などを持つ複雑なハードウェアとなっている. ここでは PS3 が存在しない状態で, PowerPC と SPU からなる Cell を CbC で記述する.

† 琉球大学理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University to the Ryukyus.
†† 琉球大学工学部情報工学科
Information Engineering,Unisity of the Ryukyus.

2. Continuation based C(CbC)

Continuation based C(以下 CbC) とは我々が提案している記述言語である.[1] 近年, オブジェクト指向という言葉に代表されるように, Java や Python などといったプログラミング言語が注目を集めている. これらは動的な適合性を中心に設計され, リフレクションなどの導入により, C などの低レベル言語より, メモリやスタックに問題が有る. このような言語は高速な応答を期待される Real-time 処理や組み込み用途に適さない.

この言語では, 関数, for 文や while 文などのループ命令はなく, 状態遷移を記述するのに, code と goto を用いる. 状態遷移を表す code は return の無い C の関数の形をしており, code から脱出するためには goto を用いる. goto は名前付き code を指定する必要が有る. CbC における goto は C のラベル付き goto とは異なっている. 以下に CbC の特徴と要求仕様を列挙する.

- ループ構造がない
- サブルーチンコールがない

- 継続が導入されている
- C に継続専用のコード単位 (code) があり, 継続 (goto) を導入した言語
- ハードウェアとスタックマシンの中間言語
- C 言語よりも下位の言語
- 状態遷移を記述できる
- Thread を実行モデルに内蔵できる
- クリティカルパスの最適化

これらの仕様はハードウェア記述とソフトウェア記述の両方を行い, C よりも精密な実行記述をするためのものである. また, SPU の一命令が一つの code セグメントで表される. また C の関数に相当する code を code セグメントと呼び, 関数の引数に相当するものをここではインターフェースと呼ぶ.

2.1 アーキテクチャに依存しないプログラム開発

CbC で SPU プログラムのシミュレートを行う際, 必ずしも特定のハードウェアで実行する必要は無い. 現在 CbC コンパイラは MIPS, PowerPC, IA32, ARM 用の CbC コンパイラが実装されている. これらのアーキテクチャ上で開発できるというメリットが有る.

3. 開発段階にそったシミュレータ実装手法

シミュレータの開発手法として C による実装などがある.[3] シミュレーションするターゲットはアセンブラレベルのものとなる. この手法はアセンブラプログラミングに置けるバグを取り除くことも可能となる. また開発段階の設計手法としてここでは 3 種類の実装方法を挙げているが, これらはよりハードウェアに近いレベルになっていく. ここでは CbC により, シミュレータを作成し, パフォーマンスや並列度の評価を行う.

実装方法はアセンブラ命令を一つの code セグメントに直すものである.
実装 1 は CbC の特性を踏まえ, なおかつアセンブラに近い使用法になっている. 一つの code セグメントが終了すると, 次の code セグメントへジャンプという形式である. (図 1) これは明示的にプログラムカウンタを必要としない.

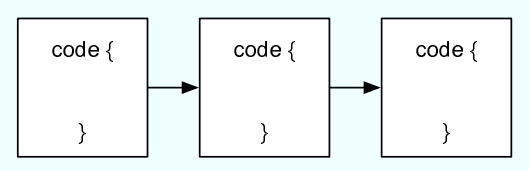

図 1 実装 1

実装 2 では実装 1 よりもよりアセンブラに近くなる.

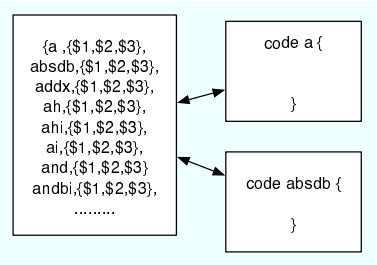

図 2 実装 2

アセンブラそのもの, もしくはアセンブラに相当するものをプログラム中に記述しておく. そのアセンブラに相当するものを一つずつ読み出し, そのオペコードに相当する code セグメントへジャンプする. (図 2)

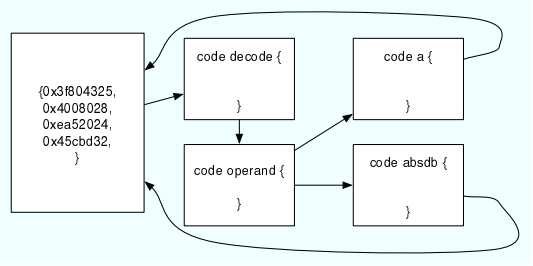

図 3 実装 3

実装 3 ではハードウェア記述でいうデコーダに相当するものを CbC で記述する手法である. 命令は全てビットパターンで表され, ビットパターンからオペコードとオペランドをデコードする. これは実際の CPU がやっていることでも有る. 実装 1, 実装 2 の方法と比べて, よりハードウェアに近いレベルとなる.(図 3)

4. PlayStation3 の構造

ここでは, ターゲットとなる PlayStaion3 の構造に付いて説明する. PlayStation3 は, Cell と呼ばれるマルチコア CPU と Grahpics Processing Unit I/0 などから構成されている. ここでシミュレートするのは Cell である. Cell は IBM の 64 ビット Power アーキ

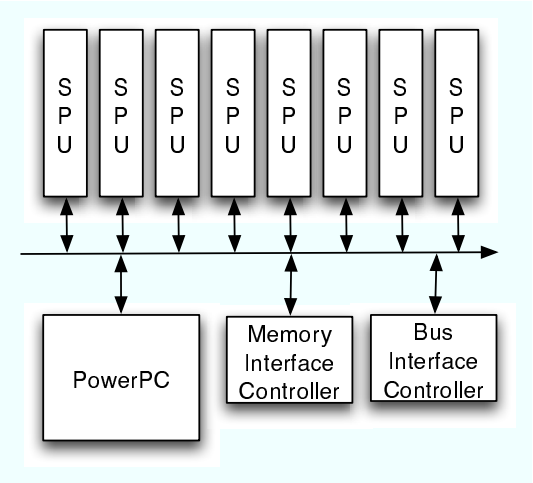

図 4 cell のイメージ図

テクチャをベースに開発された Power Processor Element(以下 PPE) と呼ばれる汎用のプロセッサコア一つと, 8 つのマルチメディア処理に適した Synergistic Processor Element(以下 SPE) と呼ばれるものから成り立っていて, それぞれはリングバスによって, 接続されている. (図 4) PPE は Cell の中のコントロール用のプロセッサコアと考えることができる. PPE は SPE をスレッドとして扱うことができ，PPE 自身も 2 つのスレッドを実行できる. しかし，8 個有る SPE は全部使うのではなく, 6 個しか使わない. また PPE にはキャッシュが 2 つ内蔵されている. PPE の仕事は OS を動作させること, SPE の動作を制御することである.

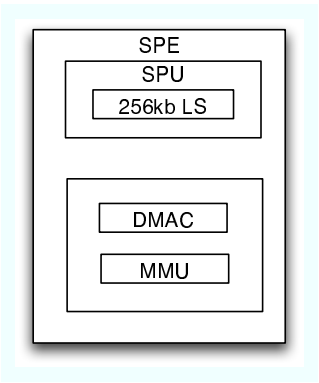

図 5 SPE のイメージ図

SPE はまったく新しい命令セットで, 128 個の 128 ビット汎用レジスタと 256kb の内部メモリを備え, Single Instruction Mutiply Data(SIMD) 構成になっている. またこれらのデータ転送は DMA で行われ, CPU に負担をかけない. 内部メモリに 256KB, DMA Controller, Memory Management Unit が内蔵されている. これらは Cell のベクトル演算などを担当する. それぞれの SPU は 4 個の浮動小数点演算ユニットと 2 個の整数演算ユニットがあり, これらは並列に動作することができる. また VU の Upper Unit や Lower Unit のように 2 命令同時発光型となっているが, 現在その詳細が記述されたドキュメントは発表されていない. そこで, ここではエミュレータをターゲットとしている.

4.1 エミュレータ

現在 IBM や SCEI やバルセロナ大学スーパーコンピューティングセンターが共同で開発段階李に置けるエミュレータ環境を提供している. そのエミュレータ環境は PS3 に搭載されるライブラリなども提供されているので, Cell プログラムを記述, コンパイル, 動かすことが可能となる. たとえエミュレータが変化しようが本研究に置けるシミュレータではすぐに対応できるというメリットが有る.

5. CbC による Cell の記述

ここで, さきほどのべたように CbC での実装方法として 3 種類挙げる.

例えば

ilhu $3, -32768

というアセンブラ記述があるとする. ilhu は Immediate Load Halfword Upper という命令で, i16 フィールドの値をワードの左端 16 ビットに置き, ワードの残りビットを 0 に設定すると言う命令である. ilhu というアセンブラ命令は次のような動作をする.

$t = i16||0x0000$
$RT[0:3] = t$
$RT[4:7] = t$
$RT[8:11] = t$
$RT[12:15] = t$

この例題を CbC に書き直すと, 次のようになる.

```
code ilhu() {
        int i16;
        i16 = -32768;
        i16 = i16 << 16;
        b. i3[0] = i16;
        b. i3[1] = i16;
```

```
        b. i3[2] = i16;
        b. i3[3] = i16;
        goto wrch7();
}
```

5.1 実 装 1

一番目の方法ではこれをすべて code に記述し, 直接的に goto する方法である.

```
code ilhu() {
        . . . . . .
        goto wrch7();
}

code wrch7() {
        . . . . . .
        goto rdch();
}

code rdch() {
        . . . . . .
        goto exit0(0), env;
}
```

code セグメント内の最後の goto で次の命令先にジャンプする. このように一つの code セグメントにつき, 一つのアセンブラ命令を記述していくことによって, シミュレートが可能となる. ここでは wrch7 という code セグメントにジャンプしている. これは次のジャンプ先が即座に分かり, 各 code セグメントでどのレジスタにたいしての操作かもわかりやすい. ここでのプログラム中では b. i3 が$3 に対応するレジスタになる.

5.2 実 装 2

2 番目の方法は

```
typedef struct {
        code (*opcode)();
        operand operand;
} instruction, *instruction_ptr;
```

構造体を用意して,

```
instruction prog[] = {
        {ilhu, {r3, -32768}}
}
```

というふうに命令を全て構造体に入れておく.

```
code spu2_ilhu(operand op) {
        op. rs = (int)op. rs << 16;
        op. rt[0] = op. rs;
        op. rt[1] = op. rs;
        op. rt[2] = op. rs;
        op. rt[3] = op. rs;
}
```

とポインターで値を代入するほうほうがある. 構造体 operand は対応するレジスタのポインターになる. この方法は各命令の動作記述が明確になる.

5.3 実 装 3

3 番目の方法はビットパターンからオペコードとオペランドをデコードする方法が挙げられる.

```
inst prog[] = {
        {0x31a00907}, {1}
};

code decode(instruction ins) {
    int opcode;
    if(prog[pc]. type == 1) {
        opcode = prog[pc]. ope & 0xfe000000;
        opcode = opcode >> 25;
        ins. opcode = t1[opcode]. opcode;
    }
    . . . . . .
    goto decode_operand(ins);
}

code decode_operand(ins) {
     int rt, rs, ra;
     if(prog[pc]. type == 1) {
         rt = prog[pc]. ope & 0x01ffff80;
         rt = rt >> 7;
         ins. operand. rt = rt;
         rs = prog[pc++]. ope & 0x7f;
         ins. operand. rs = rs;
     }
     . . . . . .
     goto ins. opcode(ins);
}
```

というふうに構造体に定義し, ビットパターンからオペコードとオペランドをデコードする code セグメン

トを用意する．オペコードはタイプに応じて, 配列に宣言されている．ここでは先ほどと同じ様に各命令に対する code セグメントが一つずつ用意されている．

5.4 SPU の並列実行

先ほど説明した 3 種類の方法で, SPU を並列実行させる方法は Schedular を用いることである．

```
typedef struct {
  code (*s1)();
  code (*s2)();
  code (*s3)();
  code (*s4)();
  code (*s5)();
}sch;
```

というような構造体を用意し，各 code には各 SPU に置ける次の命令が入っている．

```
code spu1_ila(sch s, channel c) {
   ..........
   s. s1_next = rotqbyi(s, c);
   goto schedular(s, c);
}
```

ここでは ila 命令の次は rotqbyi になっている．
schedular は

```
code schedular(sch s, channel c) {
   if(s. s1==exit0 && s. s2==exit0........)
        goto exit0(0), env;
    else if(count%6 == 0 && s. s1 != exit0) {
        count++;
        goto s. s1(s, c);
    } else if(count%6 == 0 && s. s1 == exit0) {
        count++;
        goto schedular(s, c);
    } else if(count%6 == 1 && s. s2 != exit0) {
        count++;
        goto s. s2(s, c);
    }
    .......
}
```

これは一番最初に示した方法の Schedular になる．2 番目に示した方法と 3 番目に示した方法も大体同様となる．2 番目の方法では各 SPU が命令を保持していて, Schedular でカウントして挙げればよい．3 番目の方法では 2 番目と同様に Schedular でカウントすればよい．

6. 実装のまとめ

- 実装 1 の方法では 1 つの命令が 1 つのコードセグメントに対応するので, コードセグメントを見れば動作が一目瞭然
- 実装 1 の方法では同じ動作記述でもレジスタが変わるだけでコードセグメントが一つ増えるため, ソースが大きくなる．
- 実装 2 では一つのオペコードに付き, 一つのコードセグメントが対応するので, コード量は増えない
- 実装 2 では構造体を見れば, どういう順序でアセンブラが実装されているかわかる．
- 実装 3 , 実装 2 , 実装 1 の順にハードウェアシミュレーションにより近くなっている．

7. Cell シミュレーションに置ける CbC の利点

C 言語互換 CbC は C 言語の下位互換言語なので, CbC による Cell プログラムの記述は C 言語に近い形式であり，Cell の特殊なアセンブラを知らなくても, 動作を理解することができる．これにより, アーキテクチャによって仕様が異なるアセンブラを理解する手間が省ける．これではアセンブラのマニュアルとして利用することが可能となる．

スタックの廃絶 CbC は C 言語などに有る通常の関数呼び出しは行われずレジスタ上に必要なデータが保持されるので, スタックの処理が必要でなくなる．

実行時のチェック PS3 のアセンブラ命令の実行時に値を出力したりして, デバッガとして使用可能である．

CbC コードの自動生成 アセンブラから CbC のコード生成が可能である．特に実装 2 では構造体の配列にアセンブラを代入さえすれば, あとは対応する命令があれば, CbC コードができたことになる．

8. 各実装方法に置ける実行時間

アセンブラ命令を CbC で記述する例題として, IBM

らが提供しているエミュレータ上で実行することが可能な simpleDMA という例題を用いた．simpleDMA のアセンブラソースは次のようになっている．先ほども述べたが，2 命令同時発光型となるので，このアセンブラとは異なることが分かっている．

```
.file "simpleDMA_spu.c"
.text
.align 3
.global main
.type main, @function
main:
ila $9,cb
rotqbyi $14,$4,8
shlqbyi $13,$14,12
rotqmbyi $12,$13,-4
rotqmbii $11,$12,0
rotqmbyi $10,$11,-8
shlqbyi $7,$10,12
wrch $ch16,$9
il $8,0
wrch $ch17,$8
wrch $ch18,$7
il $6,128
wrch $ch19,$6
il $5,31
wrch $ch20,$5
il $4,64
hbr .L3,$lr
wrch $ch21,$4
ilhu $3,-32768
wrch $ch22,$3
il $2,2
wrch $ch23,$2
rdch $3,$ch24
il $3,0
nop $127
.L3:
bi $lr
.size main, .-main
.comm cb,128,128
.ident "GCC: (GNU) 3.4.1 (CELL 2.3, Jul 21 2005)"
```

ila, il, ilhu 命令はそれぞれ定数生成命令，rotqbyi, shlqbyi, rotqmbii 命令はシフトおよびローテート命令, wrch, rdch はチャネル命令を表す．チャネル命令により，外部の SPE および PPE とデータ転送を行う．定数を生成して，他の PPE もしくは，SPE にデータを転送しているだけの例題である．これらの例題をそれぞれの実装方法で動かして時間を計測してみた．マシンは Pentium3 の 1GHz のメモリ 512MB 上で動かした．

	実装 1	実装 2	実装 3	エミュ
速度	0．357377	1．290057	2．582058	12

表 1 実行速度

これらはエミュレータ以外は gettimeofday 関数を用いて計算した．計算結果は 100 万回実行した結果である．エミュレータには gettimeofday がなかったため，time コマンドを用いた．

9. ま と め

CbC は状態喘記述に適しており，その用途は幅広い．本研究では CbC による PS3 Cell の SPU アセンブラプログラムのシミュレートを 3 種類の方法で行った．CbC による SPU アセンブラプログラムは IBM らが提供しているエミュレータよりも高速なシミュレータを持つことが可能になる．また，今現在シミュレートしているのはエミュレータのシミュレートに過ぎない．実際は 2 命令発光型になるので，今現在示しているアセンブラの例題とは異なり，PS2 VU の用になるとおもわれる．実際にどのようになるか分かったときにそれに対応したシミュレートに直す必要性が出てくる．またパイプラインハザードなども考慮に入れなければならない．また，他にこのようなシステムを記述する方法として VHDL に代表されるハードウェア記述言語がある．これらのハードウェア記述言語よりも元のアプリケーションを記述する言語との親和性が良い．

参 考 文 献

1) 河野真治 継続を持つ C の下位言語によるシステム記述.
日本ソフトウェア科学会第 17 会大会
2) 佐渡山 陽, 河野 真治. Continuation based C による PS2 Vector Unit のシミュレーション,
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.nnue 2002.
3) 河野 真治. 継続を基本とするプログラム単位を用いた組み込みシステム開発, 組み込みソフトウェア工学シンポジウム,Oct 2003
4) Sony Cell Broadband Engine 公開情報

http://cell.scei.co.jp/j_download.html
5) ソニー,IBM, 東芝
A Streaming Processing Unit for a CELL Processor