Continuation Based C による PS2 Vector Unit のシミュレーション

河野 真治
科学技術振興事業団さきがけ研究 21(機能と構成)

佐渡山 陽
琉球大学大学院理工学研究科

概要

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) を提案している。CbC は、デバイスドライバや OS の記述、また、最適化自体を記述するのに適しているが、ハードウェア自体の記述も可能である。ここでは、PS2 に内蔵された DSP の Vector Unit (VU) の記述を行い、その有効性を示す。CbC で記述された VU の記述は、実際に PS2 上で動作させることが出来る。これにより、内蔵ハードウェアの性能評価を実機上で内蔵ハードウェア抜きで行うことができる。実際の実行時間は、内蔵ハードウェアの 1/1700 程度であった。

The simulation of PS2 Vector Unit by Continuation Based C

Shinji Kono
Japan Science and Technology Corporation

Akira Sadoyama
University of the Ryukyu

Outline

We are demonstrating Continuation based C (CbC), which is a low level language of C. CbC is suitable for describing state machine such as device driver, operating system or optimization itself. It is also possible to describe hardware. Here we show a description of PS2's VU which is embedded DSP. The CbC description actually run on PS2. In this way, we can evaluate possible embedding hardware on the target machine. Actual execution speed is 1/1700 of real hardware.

1 CbCによる組み込みハードウェア設計の評価手法

近年のアプリケーションは、既存の汎用PCに単にソフトウェアを載せるという形よりも、組み込まれた特殊なハードウェアと供に構築されることが多い。例えば、ウィンドウズアプリケーションは、高速なビデオチップを想定して作られている。また、ハードディスクやネットワークとのやりとりも専用のハードウェアで行われることが多い。このようなハードウェアと協調するソフトウェアの構成は、ハードウェア側からは、ハードウェア・ソフトウェア協調合成のような形で提案されているが、ソフトウェア側からのサポートは、システムコールやライブラリコールのレベルでしか提供されていない。

ここでは、通常のプログラミング言語よりも下位に位置する言語、Continuation Based C (以下 CbC) を用いて、組み込みハードウェアとソフトウェアの協調動作を表現する方法を提案する。

表現するターゲットとしては、SCEI の PlayStation2 (以下 PS2) を採用する。PS2 では、Linux を走らせることができ、ユーザによるプログラミングが可能になっている。PS2 は、MIPS CPU を中心として、VU と呼ばれる二つの DSP ユニットと描画エンジンである GS、その他、旧 PlayStation との互換性をとるための IO プロセッサなどを持つ極めて複雑なハードウェアとなっている。より単純な構造を持つ PlayStation では、C と、そのライブラリコールのような形でプログラムすることが可能であったが、PS2 では、VU を積極的に使用しない限り、その性能を生かすことは出来ない。

ここでは、VUが存在しない状態でVUを用いたプログラミングの評価を行う状況を想定する。CbCによるVUの記述を行い、それを実機上のMIPSコアで動作させ、実行クロック数やパイプラインハザードの状況などを調べることが出来た。CbCによるVUの記述の実行は実機の1/1700程度であった。

2 Continuation based C

2.1 CbC

CbCとは、状態遷移単位での処理の記述を行うCの下位互換言語である(図1参照)。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の関数の形をしている。codeは、Cのブロック(ブレースで囲まれたコード区分)である。そして、そのブロックから出るためには、goto文で何処かの状態遷移へ飛ぶか、もしくは、ブロックの最後で、そのまま下へ抜ける形となる。下へ抜けるとは、現状態遷移の下に記述されているcode文(状態遷移)へ移行する事である。goto文の行き先には、名前付きcodeか、または、それをさす変数である。

CbCのcodeとgotoには、引数があり、処理に必要なデータや状態遷移の管理に必要な情報を持ち運ぶ。通常は、状態を表す単一の構造体を引数に持つ。VUのCbCによる記述では、この構造体は、VUのレジスタの集合である。

VUの一命令が一つのcodeセグメントで表される。codeセグメント内では、VUの状態、つまり、レジスタの内容を調べ、命令の動作にしたがって、レジスタの内容を変更し、次のVU命令をプログラムカウンタから取得して、次のcodeセグメントへgotoする。つまり、VUは、状態を表す構造体と、code(状態遷移)の集まりからなる記述される。

Cの関数呼出しと異なり、インタフェースに相当するcodeセグメントの構造体が等しければ、codeの呼出しの手間は、ジャンプ命令1命令だけである。また、呼出しの度にスタックを消費することもないので、シミュレーションなどの記述を実行する際の実行時ワーキングセットを小さくできることが特徴である。

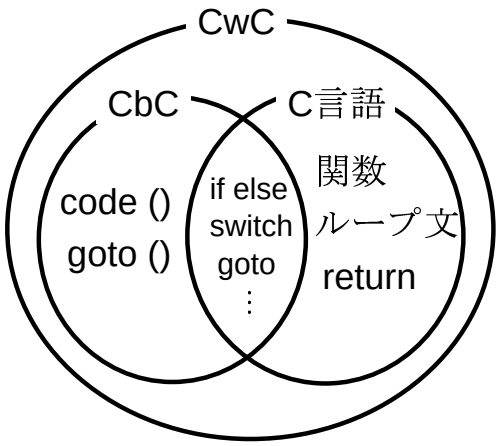

図 1: 言語互換相図

2.2 ハードウェア記述

CbCは、状態遷移に沿って処理を1つ1つ記述して行くことにより、ハードウェア記述言語としても利用できる。実際に実行して、そのハードウェアをシミュレーションできるので、その結果をから、アーキテクチャ自体のデバッグや改良をする事ができる。

2.3 高速・高性能マシンによるプログラム開発

CbCはプログラムのシミュレーションを行う際、必ずしも特定のハードウェア上で実行する必要は無い。ゲーム開発に留まらず、処理能力の低いハードウェアやインターフェースの悪いハードウェアに対する開発を行う場合、CbCを用いて高速・高性能なマシン上でシミュレーションを行い開発を進めれば、開発にかかる時間を縮める事が出来る。プログラムの実行に対するハードウェアの状態遷移

の記述の詳細度を選択することにより、低速で正確なシミュレーションや、精度は低いが、より高速なシミュレーションを選択することが可能である。

プログラムは、code 単位で分割されているので、これらの選択は、code 単位でのプログラム変換で行うことができる。

2.4 ソフトウェアとハードウェアの協調記述

CbC は、C の下位言語であるので、C のプログラムを CbC で記述することが可能である。したがって、CbC のレベルでは、ソフトウェアの記述と、CbC がシミュレーションするハードウェアの記述を混在することができる。これを、ハードウェアのシミュレーションとして、実行することが出来る。また、CbC で記述されたハードウェア部分を、VU の命令、マイクロコード、あるいは、ハードウェアにマッピングするプログラム変換器、または、コンパイラを用意すれば、実際のハードウェア上で実行される記述を行っていることになる。

code セグメントによる VU の動作の記述から、VU のアセンブラを生成することは、一般的には難しい。しかし、CbC の code セグメントのインタフェースを、VU のレジスタに直接に対応させるようにすれば、対応するレジスタに対して同じ動作をする命令セットを見つけることを行えば良いので、若干、問題は簡単になる。これが、CbC のアセンブラに相当する使い方となる。これを短く CbC の VU コンパイラと呼ぶ。

3 PlayStation2 の構造

ここでは、ターゲットとなる PlayStation2 の構造に付いて説明する。PlayStation2 は、メインプロセッサである Emotion Engine と Graphics Synthesizer、IO プロセッサからなる。ここでシミュレーションするのは、Emotion Engine の一部を構成する 2 つの DSP(信号処理プロセッサ) VU (Vector Unit) である。

EE(Emotion Engine は、PlayStation2 のメインプロセッサで、300MHz の MIPS Core と 2 つの VU と呼ばれる DSP などから構成される密結合並列計算器である。(図 2 参照) 全ての演算処理を EE で行い、その結果を Graphics Synthesizer と呼ばれるグラフィックプロセッサへと送ることにより、映像を映し出す事ができる。また、ユニット・プロセッサ間のデータの転送は、DMA で行われ、CPU に負担をかけない。

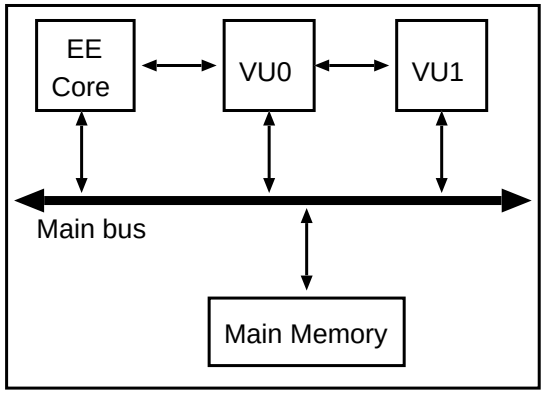

図 2: Emotion Engine 略図

VPU は、EE の一部であり、ベクトル演算処理などと担当する。その中心は、VU0, VU1 の二つの DSP である。VU は、MIPS コアとは独立して動作するが、VU0 は、コプロセッサとしても使用することが出来る。VPU には、DMA で送られて来たデータを VPU 内部のメモリへ送ったり、VU 自体の制御を行う VIF がある。

一つの VU は、さらに、Upper Execution Unit と Lower Execution Unit の二つに分かれ、それぞれの動作を 1 命令の中で独立に指定できる。Upper 命令では、浮動小数点の加算、減算、乗算、積和計算が行われる。Lower 命令は、より普通の CPU に近く、レジスタ間転送、ジャンプなどの制御命令を含む。以下に、VU アセンブラの例を示す。左が Upper の命令、右が Lower の命令である。

```
NOP  LQI.xyzw VF06, (VI01++)
```

```
NOP  LQI.xyzw VF07, (VI01++)
MULAx.xyzw ACC, VF04, VF08x  LQI.xyzw VF09, (VI02++)
MADDAy.xyzw ACC, VF05, VF08y NOP
```

最初の命令では、左の Upper Unit はなにもしていない。右の LQI は、Lower Unit で実行され、レジスタ VF06 の xyzw フィールドに、VI01 が指し示すアドレスのデータを読み込み、さらに VI01 をインクリメントしている。3 番目の命令では、Upper Unit は、レジスタ VF04 と VF08 の x フィールドの値をかけて、アキュームレータ ACC に格納している。4 番目の命令では、Lower Unit は、パイプラインハザードを避けるために、NOP(No Operation) を実行している。

このように、VU のアセンブラは極めて複雑な構造をしており、膨大なマニュアルを読まない限り、その動作を理解することはできない。また、VU の複雑なパイプライン実行を考慮しつつプログラミングを行うことが必須である。しかし、これらの機能を生かすことにより、汎用 CPU の数倍の計算能力を発揮することが出来る。

4 CbC による VU の記述

前章で説明した Lower Unit の LQI 命令の実行は、以下のように CbC の code セグメントとして記述される。

```
code lower_LQI(REGISTERS *regs,
  PARAMETERS *params, HAZARD_CHECK *hazard)
{
  if (++hazard->F_add_point == FMAC_STALL)
    hazard->F_add_point -= FMAC_STALL;
    ...
```

ここで、regs は、VU のレジスタの内容である。params は、LQI 命令のオペランドが入る。hazard は、パイプラインハザードチェック用の状態である。

命令の記述の大半は、ハザードのチェックなどに費されるが、実際の実行は、以下のように、レジスタ構造体への値の変更の記述となる。

```
/* LQI */
if (params->ft != hazard->void_VF) {
  if ((params->dest & 8) != 0)
    regs->VF[params->ft][0] =
       VU_MEM[regs->VI[params->is]][0];
  if ((params->dest & 4) != 0)
    regs->VF[params->ft][1] =
       VU_MEM[regs->VI[params->is]][1];
  if ((params->dest & 2) != 0)
    regs->VF[params->ft][2] =
       VU_MEM[regs->VI[params->is]][2];
  if ((params->dest & 1) != 0)
    regs->VF[params->ft][3] =
       VU_MEM[regs->VI[params->is]][3];
}
regs->VI[params->is]++;
```

このような記述が、Upper Unit と Lower Unit の両方に対して存在する。この記述は、VU の命令の実行の詳細を記述していると考えても良いし、VU のハードウェアの記述と考えても良い。また、実際に動作するアプリケーションの動作の一部を記述しているとも言える。

5 VU プログラミングにおける CbC の利点

[C 言語との互換性]

CbC は、C 言語の下位互換言語なので、CbC による VU プログラムの記述は、プログラマーの視点から見て C 言語に近い形式であり、VU 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出来る。CbC のコードは C と殆ど同じなので、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。また、ISP の様に VU の命令セットの記述、あるいは、アセンブラの機械可読なマニュアルとして使うことが出来る。

[実行時のチェック]

また、PlayStation2 の VU プログラミングにおいては、Upper Execution Unit と Lower Execution Unit の並列処理を考慮する必要がある。CbC の記述に、並列処理で生じるパイプラインハザードなどの命令間の干渉のチェックを入れることが可能である。

[レジスタの明示]

VU のレジスタは、code セグメントのインタフェースで明示されている。CbC では code セグメントのインタフェースが一致している場合、レジスタのマッピングやメモリ上のデータ移動が必要ないので、goto 文はただ一つのジャンプ命令にコンパイルされる。VU 記述では、基本的に同じインタフェースを用いるので、効率的なコンパイルが可能である。

[スタックの廃絶]

CbC では通常の関数呼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、CbC ではスタックの処理が必要ではなくなる。また、Working set を小さくすることが可能になり、キャッシュを有効に使った高速なシミュレーションが可能になる。これは、記述をハードウェアに対応させるためにも、隠れたスタックがないことは有効である。

[最適化の記述]

CbC の記述は、インタフェースで指定されたレジスタに対して直接記述される。アセンブラレベルでの最適化はレジスタに対して行われるので、CbC は、アセンブラレベルでの最適化を記述する能力がある。クロック数を CbC の記述に含めることにより正確なクロック数のシミュレーションをすることも可能である。

C 言語では、最適化は C 言語自体に対して適用される。CbC では、CbC の記述が VU の命令と対応していれば、CbC を出力とした最適化を、CbC コンパイラとは別に、プログラム変換として記述することができる。つまり、プログラミング言語と、その最適化を分離することができる。

[CbC コードの自動生成]

アセンブラから、CbC のコードを自動生成することが可能である。逆に、CbC の記述からアセンブラを生成する事もできる。

6 CbC による VU のシミュレーション

VU のアセンブラを、CbC で記述して PS2 の実機、および、汎用計算機でシミュレーションする実験を行った。シミュレーション用の VU アセンブラプログラムには、PlayStation2 Linux Kit に収納されていた例題を使用した。例題では、VU1 において、

1. 座標系の演算
2. 透視変換
3. 光源計算
4. テクスチャマッピング

等の処理を CPU と並列で処理させる (マイクロモードで作動させる) ことにより、全体的な処理速度を上げている。

シミュレーションでは、本来 VU で行っている処理を CbC によって記述し、それを Core において実行させている。

もっと速い PC 上ではなく、PlayStation2 上でシミュレーションを行うことにより、処理した結果を実機の Graphics Synthesizer へと送ることができる。これにより、シミュレーション結果の正しさを、実際の描画によって確かめることが可能である。シミュレーションが正しければ、例題と同じ画像が出力されるはずである。ただし、分配していた処理を Core だけで行うので、当然ながら処理速度は遅くなる。

7 C による CbC の実装

CbC の完全なコンパイラが、まだ完成していなかったので、C を用いて CbC の実装を行い、それでシミュレーションを行う事にした (図 3)。

code は、int 型の関数を呼び出すことで実装した。しかし、goto 文の様に、関数から関数へと return せずに次々と飛んで行くと、スタックがどんどんたまっていき、segmentation fault を起こしてしまう。

そこで、関数のポインタを用いる方法を考

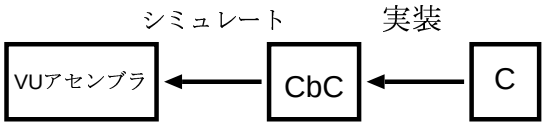

図 3: 多段シミュレーション

え、Dispatcher を作成した。指定された code を、C の関数として呼び出す。return 時に、次の状態遷移を記述した関数のアドレスを返すことで、goto 文を使用しているかのように、次々と状態遷移間の移動が行える（図 4)。

以下に具体的な C の記述を示す。ptr が、int 型関数のポインタである。次の状態遷移を示す ptr が NULL になるまで、ループして次の状態遷移を呼び出す。

■ C による状態遷移の実装

```
int (*ptr)(REGISTERS *regs);
int state1(REGISTERS *regs);
int state2(REGISTERS *regs);

int state1(REGISTERS *regs)
{
  状態の判定;
  状態に対応してパラメーターの
  値を変化;
  return((int)state2);
}

int state2(REGISTERS *regs)
{
  状態の判定;
  状態に対応してパラメーターの
  値を変化;
  return((int)state1);
}

int trans_ctrl(REGISTERS *regs)
{
  (int)ptr = (int)state1;
  while(ptr) {
    (int)ptr =
        (*ptr)(regs,params,hazard);
  }
}
```

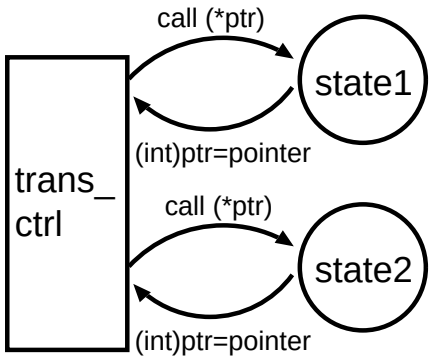

図 4: goto 文の仕様

8 CbC におけるアセンブラの記述

アセンブラの命令を CbC で実現するために、サンプルプログラムをシミュレーションするために必要なアセンブラ命令を、全て CbC の記述へと書き直した。

引数は、状態遷移間で受け渡すデータ群を、以下に示す 3 つのグループに分けて定義した。CbC によるアセンブラプログラムのシミュレーションは、これらの構造体に含まれる情報を基に実行される。

1. REGISTERS
   VU が持っている全てのレジスタを表す変数群
2. PARAMETERS
   アセンブラ命令が必要とするパラメーターと、次に進むべき状態遷移へのアドレスを持つ変数群
3. HAZARD_CHECK
   ハザードやエラーに関する処理に必要な変数群

また、書き直した各 CbC によるアセンブラ記述に、そのアセンブラ命令を実行した時に起きうるハザードの判定を行い、ハザードが起きた時には、その遅延をクロックの値に反映させるプログラムも付加した。

9　実行時間と実行クロック数

シミュレーションにかかる時間を、gettimeofday 関数を用いて計測した。計測結果は、100 回実行した平均である。ハザード検出を除けば、さらに高速になると考えられる。

また、比較のために、同様のシミュレーションを、1.6GHz の CPU を持つ汎用 PC で行った。

PlayStation2

実行したクロック数	823
平均実行時間	0.004371 秒
1 クロックの実行時間	5.311 μ sec
PlayStation2 との速度倍率	1/1700 倍

汎用 PC

実行したクロック数	823
平均実行時間	0.000337 秒
1 クロックの実行時間	0.409 μ sec
PlayStation2 との速度倍率	1/130 倍

10　比較と評価

PS2 の様な汎用 CPU と専用ハードウェアという組合せは、より低いクロックで処理能力を上げることができるという利点がある。このようなシステムを記述する方法としては、以下のような方法がある。

- 専用ハードウェアを VHDL などで記述する
- システム全体を C などでシミュレーションする
- シミュレーション専用ハードウェアを使う

CbC は C に近い言語なので、VHDL による記述よりも、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。この実験でも、実機上で、実際のシミュレーションを行うことが可能なことを示すことが出来た。このようなことは、VHDL などでは試作ハードウェアを作成しなければ不可能である。

システム全体を C で記述する手間は、CbC の手法とほぼ同等である。しかし、CbC では、インタフェースなどの共通化により、より高速なシミュレーションを行うことができる。また、インタフェースがレジスタを直接に持っているので、より、ハードウェアに密接に結び付いた記述が可能である。C で、このような記述を行うと、関数呼出し時のオーバーヘッドが実用的でないほど大きくなってしまう。

シミュレーション専用ハードウェアを使用することにより、VHDL などの記述を含む実行を、実機の数分の一のスピードで、直接的に行うことができる。しかし、この場合は、汎用 CPU 部分をシミュレーションエンジンとどのように結び付けるかが問題となる。PS2 のような特殊なハードウェアの場合は非常に困難である。また、シミュレーション専用ハードウェアは高価であり、短期間でちんぷ化しやすい。CbC では、実機上での動作以外にも、汎用 PC を使ったシミュレーションが可能なので、このような問題は生じにくい。

単に C で書くより、多くの記述が必要である。また、それらの記述は、冗長な部分が多い。

11　まとめ

CbC は、状態遷移記述に適しており、その用途は幅広い。本研究では、C による CbC の実装、実装した CbC による VU アセンブラプログラムのシミュレーションを行った。今後の課題として次のものがあげられる。

1. VU 用 CbC コンンパイラの作成
2. CbC を実際に適用した VU プログラムの作成
3. 本動作記述の詳細化

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 回大会論文集, September, 2000(in Japanese)

[2] 河野真治, 島袋仁:C with Continuation と、その PlayStation への応用, 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), May, 2000 (in Japanese)

[3] PlaySation2 Linux Kit
付属マニュアル

[4] 長谷川祐恭:VHDL によるハードウェア設計入門