SN SYSTEMS
Sony Computer Entertainment Group

ホワイト ペーパー:SN008

PLAYSTATION®3 開発用 SNC PPU ツールチェーンの活用法

SN Systems Limited
バージョン:1.0
2009 年 02 月 09 日

目次

1: はじめに

本ドキュメントでは、主にコンパイラ (SNC) とリンカーから成る SNC PPU ツールチェーンを、Sony PLAYSTATION®3 (PS3) のゲーム開発において最大限に活用する方法の概要が説明されます。また、PS3 のツール使用や PS3 PPU プロセッサのコーディングにおけるヒントやアドバイスも記載されています。

記載の機能及び用いられているサンプルは SNC 270.1 からのものですが、明示的な注意書きが無い限り、以前の SNC バージョンでもサンプルは動作します。しかし、生成されるコードはここでの SNC 270.1 ベースのサンプルとは異なります。

コンパイラやリンカーはもちろん、SNC PPU ツールチェーンにはアセンブラ/逆アセンブラ、その他にバイナリ ツールも含まれています。SNC PPU ツールチェーンを構成するリンカーやその他ツールはすべて、別ゲームのプラットホームにおいて開発および導入されてきた SN Systems の独自の技術をベースとしています。

SNC は、PS3 PPU プロセッサ用の ISO 準拠 C/C++ コンパイラです。このコンパイラは、過去 20 年にわたって開発されてきた Apogee コンパイラ技術から派生および開発されたものです。Apogee コンパイラ技術は成熟した技術で、MIPS、Arm/Thumb、PowerPC など多くのターゲットに移植されてきました。このコンパイラでは、C++ 構文解析や意味解析に対し、業界標準である EDG (Edison Design Group) のフロントエンドが使用されています。

このコンパイラでは、以下の点が開発および強化されてきました。

- PowerPC コードの生成は、C++ プログラミング ゲーム開発用に最適化されています。たとえば、ゲームのパフォーマンスにはクラス メンバ関数のインライン化が不可欠ですが、過剰なインライン化は結果としてコードの肥大化を招くため、このような状況は避けなければなりません。このコンパイラのインライン化方法は、これら 2 つのゴールをうまく両立することに重点を置いて、開発されてきました。
- PowerPC のベクター演算機能には、特に焦点が当てられています。
- 一般的な PS3 ゲーム プロジェクトは、ソース ファイル数やコード行数という点において、非常に巨大なソフトウェア プロジェクトです。このコンパイラは、このような巨大ソフトウェア プロジェクトの処理に最適となるように構築されています。

コンパイラのオプティマイザ (最適化機能) には、標準的なコンパイラにおける最適化 (グローバル/ローカル レジスタ割当てや、命令スケジューリングなど) が含まれている一方、最新の SSA (Static Single Assignment) フォーマットに基づいた各種最適化 (PowerPC 特有の一連の最適化を含む) も含まれています。このオプティマイザは、特有のコード ベンチーマークをターゲットとしたものではなく、実際のゲーム コード パフォーマンスに基づいて開発および調整されています。PS3 プラットホームにおいてコード サイズとパフォーマンス間のバランスは非常に重要です。そのため、SNC ではそのバランスを取りながら最適化を行います。

ツールチェーンの重要な特徴として、コンパイラとリンカーのオブジェクト ファイルが共に PPU ABI に完全適合するため、GCC で生成されたオブジェクト ファイルと自由に交換できることがあります。

SN ツールチェーンのもう 1 つの主要コンポーネントは、リンカーです。コンパイラと同様に、リンカーも実際のゲーム コードを使用して開発および最適化されているため、PS3 のゲーム開発に最適となっています。リンカーには、PS3 のサイズ制限内に収まるように、未使用または重複セクションをストリッピングすることにより、コード サイズやデバッグ データのサイズを削減するオプションが含まれています。また、リンカーでは、テンプレートなどの C++ の書式によって肥大化したコードに存在する多数の共通ソースの最適化も行われます。

本ドキュメントの後続部分では、SN ツールチェーンを使用したプロジェクトのビルド手順について説明します。具体的には以下のステップがあります。

- 分散型ビルド ファームやプリコンパイル済みヘッダ ファイルを使用した、ビルド環境のセットアップと、ビルド速度の最適化。
- パフォーマンス問題の原因を探るためのパフォーマンスやサイズの計測。
- SNC のパフォーマンス調整。ここでは、主にパフォーマンスに重点を置きますが、サイズの問題についても一部説明します。一般的なコンパイラ オプションと PPU 特有オプションの両方を取り上げます。
- PS3 PPU Architecture における具体的なコーディング問題。これらは多くの場合ツールチェーンに依存するものではなく、PS3 PPU Architecure に依存するものです。
- リンカーでの最適化。これは主にコード サイズの最適化になります。

参考文書

ツールチェーン リリースのヘルプ フォルダには、以下の SNC PPU ツールチェーン ドキュメントが含まれています。

ProDG_PS3_Compiler：ProDG コンパイラ ドキュメント

ProDG_PS3_Linker：ProDG リンカー ドキュメント

ProDG_PS3_Utilities：ビルド ユーティリティ、ps3name.exe、ps3snarl.exe、ps3bin.exe に関するドキュメント

2: ビルド環境

このセクションでは、プロジェクトのビルド速度に影響するファクターをいくつか取り上げます。

ホスト マシンの条件

ホスト メモリ

マシンに搭載されている物理メモリが多いほど、パフォーマンスも向上します。SNC PPU ツール チェーンでは、大きな物理メモリ アドレスと仮想メモリ アドレスに対処できます。このためツールでは、/3gb ブート オプションを使用することにより、64 ビット版 Windows プラットホームの場合は最高 4 GB まで、および 32 ビット版 Windows プラットホームの場合は最高 3 GB までに対応できます。Microsoft のブート設定データに関する詳細は、MSDN Web サイトを参照してください。[1]

リンク プロセスは、ほぼすべてが I/O バウンドです。RAM が足りない場合は、結果として仮想メモリ スワップ ファイルの使用が増加し、リンクに大きな影響を与える恐れがあります。ホスト マシンには、リンクで含まれるオブジェクトファイルとライブラリ ファイルの合計サイズに対し、少なくとも 1.5 倍のメモリを搭載することを推奨します。通常、PS3 ゲームの完全リンクには 2 GB が妥当だと考えられます。[2]

ホスト プロセッサ

SNC は単一スレッドのプロセスですが、より大きなプロジェクトでは、複数ソースの並行ビルドをうまく活用することができます。ローカル マシンにおける並行ビルドの数は、VSI や SN-DBS などのビルド システムで容易に設定できます。ただし並行ビルドの使用は、メモリ競合のため常に有益であるとは限りません。ホスト マシンには、スワッピングなく並行コンパイルを可能にする十分なメモリが必要とされます。通常は、スワッピングを引き起こす並行ビルドよりも、シリアル化されたビルドの方が好ましいと言えます。

Visual Studio と VSI

SN ツールチェーンは、SN Visual Studio Integration (VSI) を使用することにより、Visual Studio から操作できます。GCC ベースの既存プロジェクトを移植する場合は、VSI ツール addsnconfig.exe を使用することによって SNC を既存プロジェクトに追加できます。

ビルド時間を短縮させるには、静的ライブラリをなるべく使用しないことをお勧めします。VSI では個別リンクのみを実行できますが、同時にこれは、次のプロジェクトのビルドに進む前に、アーカイブ リンクで待機している並行/分散型ビルドにおけるストールを引き起こす恐れもあります。また、静的アーカイブの過剰使用もリンク時間のパフォーマンスに多大な影響を与えます。これは、適切なオブジェクト ファイルがリンクに確実に組み込まれるようにするため、リンカーでライブラリのスキャンを繰り返し実行する必要があるためです。静的アーカイブは時々ファイル リストとしてのみ使用され、アーカイブ ユーティリティによって実行される処理自体は必要とされないことがあります。

---

[1] http://msdn2.microsoft.com/en-us/library/ms791558.aspx

[2] SN リンカー マニュアル

SN-DBS

SN-DBS では、複数マシンおよびマルチコア マシンにおける並行コンパイルを実行できます。

最高のビルドパフォーマンスを得るには、以下の点を考慮に入れる必要があります。

- プロジェクトの並行ビルドを可能にするため、プロジェクトの依存関係を削減する。
- アーカイブ リンク時間は (多くの場合は不必要な) 同期ポイントを意味するため、ストールの原因となる。

プリコンパイル済みヘッダ (PCH) ベースのプロジェクトは SN-DBS と動作するものの、PCH は分配されないため使用されない、ということに注意する必要があります。ただし一般的に、PCH ベースのプロジェクトに対しては、ローカル ビルドよりも SN-DBS の方が速くなります。

SN-DBS に関するその他の最適化方法は、SN-DBS マニュアルを参照してください。

プリコンパイル済みヘッダ ファイル

多くの場合、一連のヘッダ ファイルのリコンパイルは避けることが望ましく、これは特に多数のコード行が取り込まれ、これらを含むプライマリ ソース ファイルが相対的に小さい場合に言えます。SNC には、特定ポイントでのコンパイル ステータスのスナップショットを撮り、コンパイルの完了前にディスク ファイルにそれを書き込むメカニズムが搭載されています。また、同じソース ファイルのリコンパイル時や、同じ一連のヘッダ ファイルを含む別ファイルのコンパイル時には、「スナップショット ポイント」が認識され、対応する PCH ファイルが再使用可能であるかを検証し、それを再度読み込むこともできます。これに該当する状況下では、これによってコンパイル時間が大幅に削減されます。PCH 使用時に考慮しておくべき点としてとして、PCH ファイルが大きいこと、および PCH のメリットが失われてしまうほどの I/O トラフィック量への影響があります。

適度に大きなヘッダ ファイルの場合、PCH ファイルへの書き出しと読み込みにおいて発生する相対的オーバーヘッドは、かなり小さくなります。一般的に、これが使用されない場合でも、PCH ファイルの書き出しにはさほど負担がかかりませんが、これが使用された場合は、コンパイル速度が確実に上がります。問題として、最小約 250KB から数メガバイト以上まで、PCH ファイルがかなり大きくなる可能性が挙げられます。

このため、リコンパイル速度が向上するとはいえ、前処理指令の非統一初期シーケンスを含む任意の一連ファイルに対し、PCH 処理は正当化されるものではないと言えます。むしろ、多くのソースファイルで同じ PCH ファイルを共有されている場合に、最大の効果を発揮します。共有するほど消費されるディスク領域は少なく済みます。共有が行われれば、コンパイル時間における重大な速度向上のメリットをあきらめることなく、大きな PCH ファイルのデメリットを最小限に抑えることも可能です。ヘッダ ファイルのインクルードに際して一貫性のあるプロジェクトでは、PCH によるメリットが得られます。たとえば、プロジェクトの全主要ヘッダ ファイルをそれ自体に含む、単独のヘッダ ファイルが全ソース ファイルで使用されている場合は、コンパイル時間が劇的に短縮します。反対に、ヘッダ ファイルをアドホックな方法でインクルードするプロジェクトの場合、コンパイル時間におけるメリットは一切なく、増加することさえありえます。PCH ファイルは、常に効果があるわけではありません。

3: パフォーマンスの計測

ゲーム パフォーマンスの最適化前には、パフォーマンス的に問題のある場所を推測するのではなく、パフォーマンスを計測することがきわめて重要です。

PS3 Tuner では、ゲームパフォーマンス全体における詳細な情報を得ることができます。PS3 プラットホームのにおけるチューニングでは、SPU 処理など、多数の設計トレードオフが含まれますが、ここでは PPU パフォーマンスに重点を置きます。Tuner では、PC サンプリングを使用してホットスポット分析を実行できます。

Tuner プロジェクトのセットアップ、およびプロジェクトの PC サンプルをキャプチャする方法については、Tuner ドキュメントを参照してください。データのキャプチャ後は、下ペインにあるフレームごとのトレースは無視し、より詳細な情報が得られるグローバル データを分析するため、分割線を下にドラッグします。

左ペインには、もっとも多くサンプルされた関数のリストが表示されますが、これは必ずしもゲーム内の最大ホットスポットとは限りません。これは、PC サンプルが関数全体に対して平均化され、各関数は一般的に何百もの命令であるためです。頻繁に実行された関数は、__attribute__((hot)) でマークしてください。これが、コンパイラでのインライン化やコード配置決定における指針となります。

```
PC Sample
35% [358362] <idle>
14% [145339] <kernel>
2% [22291] DoIt::DoIt(DoItXxxxxxx const&)
 2% [18783] game::DoSomething(...) ...
 1% [15133] <process-01000300>
 1% [14325] Fn1(...) ....
 1% [12751] Fn2(...) ....
 1% [11713] <libxxx>
 1% [10494] Fn3(...) ....
 1% [9596] Fn4(...) ....
 1% [9109] memcpy
 1% [8528] Fn5(...) ...
 1% [7968] Fn6(...) ...
 1% [7893] Fn7(...) ...
 1% [7321] Fn8(...) ...
 1% [7200] Fn9(...) ...
 1% [6032] FnA(...) ...
 1% [6012] FnB(...) ...
 1% [5977] FnC(...) ...
 1% [5520] FnD(...) ...
 1% [5430] FnE(...) ...
 1% [5306] FnF(...) ...
 1% [5201] FnCallBack(...) ...
<1% [4814] __cellMSStreamSetPitch
```

この例では、時間の 35% をプロセッサのアイドル状態が占め、14% は OS 内 (大半はオーディオコード内の “sleep” コール) そして、5% はその他の様々な OS プロセス及び関数で使用されています。

したがって、CPU の約 46% がゲームの実行用に残っていることになります。

この大半は、関数 DoIt::DoIt の最初の数個の命令で使用されます。[3] この約 10% が、関数に入る前の仮想関数コール オーバーヘッドです。このゲームでは、仮想関数オーバーヘッドがフレーム時間の約 5% を占めています。

[3] このライブラリ内の関数名は、匿名で作成されています。このような形式での関数命名はお勧めしません。

この特定のグラフィック ライブラリにおいては、おそらく時間の 5% 未満が計算実行に使用され、残りはメモリの移動や関数のコールに使用されます。これは、PC 環境から移植されたゲーム エンジンの典型と言えます。ただし、物理ライブラリははるかに効率の良いコードな場合が多いです。

最も「ホット」な関数をダブルクリックすると、以下のトレースが表示されます。なお、左列の値は PC サンプル ヒット カウントです。値が 500 以上の場合は、修正の必要がある遅延の大きな箇所を意味します。これらの値はカッコ付きで表示されています。

```
          DoIt::DoIt(DoItXxxxxxx const&)
(1742)    00646530 F821FF51 stdu       r1,-0xB0(r1)
                                       # Caused by a virtual call to this function
 108      00646534 7C0802A6 mfspr      r0,lr
(627)     00646538 F80100C0 std        r0,0xC0(r1)
                                       # Caused by the mfspr latency and 64 bit
                                       # ABI stack wastage.
 109      0064653C FBE100A8 std        r31,0xA8(r1)
 152      00646540 609F0000 mr         r31,r4
  89      00646544 3C800064 lis        r4,0x64
 215      00646548 38846510 addi       r4,r4,0x6510
 188      0064654C 80BF0000 lwz        r5,0x0(r31)      # here
(545)     00646550 2C050003 cmpwi      r5,0x3
                                       # Caused by LHS on pass-by reference
                                       # parameter (r5)
 109      00646554 2C850006 cmpwi      cr1,r5,0x6
 116      00646558 2F050005 cmpwi      cr6,r5,0x5
 102      0064655C FBC100A0 std        r30,0xA0(r1)
 130      00646560 2F850004 cmpwi      cr7,r5,0x4
  92      00646564 40820054 bne        0x006465B8
  33      00646568 83DF0004 lwz        r30,0x4(r31)
  13      0064656C 83E30008 lwz        r31,0x8(r3)
  52      00646570 807F0024 lwz        r3,0x24(r31)     # here
(500)     00646574 7C03F000 cmpw       r3,r30
                                       # Caused by cache miss on previous load
                                       # (in r3)
   9      00646578 2C830000 cmpwi      cr1,r3,0x0
   9      0064657C 41820028 beq        0x006465A4
  23      00646580 4186000C beq        cr1,0x0064658C
  10      00646584 4BCED99D bl         xxxxxxxxxx
  65      00646588 60000000 nop
                                       # This code is rarely executed, but takes up
                                       # valuable instruction cache load cycles.
 123      0064658C 2C1E0000 cmpwi      r30,0x0
  12      00646590 93DF0024 stw        r30,0x24(r31)
   -      00646594 41820010 beq        0x006465A4
   4      00646598 38600001 li         r3,0x1
  26      0064659C 987E002A stb        r3,0x2A(r30)
  75      006465A0 987E0029 stb        r3,0x29(r30)
  16      006465A4 809E0024 lwz        r4,0x24(r30)     # here
   6      006465A8 3C600095 lis        r3,0x95
(957)     006465AC 88840040 lbz        r4,0x40(r4)
                                       # Caused by cache miss on previous load
                                       # (in r4)
 372      006465B0 9883ED88 stb        r4,-0x1278(r3)
   7      006465B4 480004C8 b          0x00646A7C
 203      006465B8 3CC00095 lis        r6,0x95
 399      006465BC 88C6ED88 lbz        r6,-0x1278(r6)   # here
(444)     006465C0 2C060000 cmpwi      r6,0x0
                                       # Caused by access to static variable
                                       # (in r6)
  86      006465C4 408204B8 bne        0x00646A7C
 144      006465C8 41850068 bgt        cr1,0x00646630
 199      006465CC 41860158 beq        cr1,0x00646724
 100      006465D0 419A0128 beq        cr6,0x006466F8
 241      006465D4 419E00F0 beq        cr7,0x006466C4
  48      006465D8 2C050002 cmpwi      r5,0x2
  25      006465DC 41820088 beq        0x00646664
```

```
5       006465E0 2C050001 cmpwi      r5,0x1
```

ホットスポットの要因を確定するには、まず各ホットスポットで使用されたレジスタを定義する命令を探してください。ストール要因となる命令がホットスポットとして表示されることはなく、ストールは、常にホットスポットとして表示されている命令への入力のいずれかにより発生します。

PS3 に対する最適なサイズと速度を得るには、可能な限り -Os を使用することをお勧めします。「ホット」な関数について、選択的により高いレベルの最適化を適用することも可能です。

4: 最適化設定のクイックガイド

このセクションは、ビルド プロジェクトに対して一般的に有効な最適化設定に関するクイックガイドです。最高のパフォーマンスを目標としたリリース ビルドの場合は、以下を使用します。

```
-O2 -Xfastmath=1  -Xassumecorrectsign=1
```

-O は -O2 と同等です。コード サイズが重大な要素である場合は、以下のオプションを使用できます。

```
-Os -Xfastmath=1  -Xassumecorrectsign=1
```

-Os は、インライン化を除き本質的には -O2 と同じです。

典型的なデバッグ ビルドでは、以下に示すように -Od オプションを使用します。これにより、最適化されていると同時に、デバッグも可能なコードが生成されます。

```
-Od -Xfastmath=1  -Xassumecorrectsign=1
```

リリース ビルドの場合、一般的にリンカーには以下の設定が推奨されます。

```
-strip-unused -strip-duplicates
```

これらビルドのデバッグ バージョンには、-g フラグの追加が必要です。

5: SNC で調整する一般的パフォーマンス

このセクションでは、SNC のより一般的な (PS3 に特有でない) 最適化について説明します。

全般的な最適化オプション

SNC のメイン オプティマイザは、コマンドライン スイッチ -O<n> (<n> は 0 から 3) で制御します。

- -O0 ではオプティマイザが実行されません。(-O オプションが指定されない場合はこれがデフォルト)
- 値 n が増加すると、以下に説明するように、さらなる最適化が適用されます。現在のコンパイラでは、-O2 後にさらなる最適化は追加されません (つまり -O3 は -O2 と同じ)。ただし、今後のコンパイラの開発において、さらなる最適化機能が -O3 に追加される予定です。
- -Os では、サイズの最適化が行われます (速度が犠牲になる場合もある)。
- -Od では、コード デバッグの難易度を上げるような最適化 (命令スケジューリングなど) を避けるように、コンパイラに対して命じられます。このモードは、デバッグが容易に行え、実行時パフォーマンスも妥当なコードを生成するようにデザインされています。

PS3 に対する最適なサイズと速度を得るには、可能な限り -Os を使用することをお勧めします。「ホット」な関数について、選択的により高いレベルの最適化を適用することも可能です。

- `-O0` 最適化なし (デフォルト)、インライン化なし (強制インライン化を除く)。
- `-O1` 最適化なし、インライン化を許可。
- `-O2` 妥当なコンパイル時間での最適化。
- `-O3` -O2 の最適化に加え、さらに時間のかかる最適化。
- `-Od` デバッグ可能な最適化済みコードを生成 (-g を併用する必要あり)。
- `-Os` コード サイズを増加させる恐れのある最適化を避けた、コード最適化。

エイリアス分析

エイリアス分析により、ポインタや配列アクセスを通じた更新の副次的悪影響が認識され、犠牲の大きいメモリ アクセスをコンパイラが避けられるようになるため、これは PS3 において重要な役割を果たします。SNC オプティマイザでは、最適化の実行有無を決定する際、そのエイリアス分析フェーズの結果を考慮する必要があります。潜在的なエイリアスの問題のため、結果的にオプティマイザがより保守的になることが多々あります。C99 言語仕様では、タイプに基づく一連のエイリアス規則が定義されており、これは*厳密なエイリアシング (Strict Aliasing)* と呼ばれます。一般的に、C99 規則に準拠するプログラムでは、コンパイラのエイリアス分析で問題が発生することはありません。C90 と C++03 には、C99 仕様と同様のエイリアス規則が適用されます。

2 つのポインタが同じオブジェクトやアドレスをポイントする場合、これらは互いにエイリアスであると呼ばれます。「2 つのポインタが間違いなく互いのエイリアスでない」とエイリアス分析フェーズで判

断可能な場合、オプティマイザではより積極的な多数の最適化を実行し、スケジューラではより積極的なスケジューリングが実行されます。エイリアス分析フェーズにおいて、2 つのポインタが互いのエイリアスかどうかを判断できない場合は、エイリアスであると仮定する必要があるため、オプティマイザやスケジューラによる積極的な最適化は行われません。

このため、オプティマイザに可能な限り多くの情報を提供する方法で、コードを記述することが重要です。これは、ヒントを提供するキーワード (__restrict など) に加え、特定のコーディング スタイルを使用することによって実現可能です。また、コンパイラで、コードに関するある種の仮定を行わせることも可能です (具体的には、厳密なエイリアシング規則を順守)。ただし、コードではこれら仮定を決して破ってはならず、それが守られない場合、不適切なコードが生成されることもあります。

厳密なエイリアシング

厳密なエイリアシングは、原則的に C99 言語仕様を前提としています。こう仮定することにより、コンパイラでは、エイリアス分析フェーズを「緩和」できます。これは、-Xrelaxalias=<n> スイッチを使用して制御され、n には以下のいずれかが入ります。

- 0　コードが C99 の厳密なエイリアシング規則に準拠していないと仮定。より保守的になる。
- 1　タイプ インスタンスが重複しないと仮定し、エイリアス分析を緩和。
- 2　C99 のエイリアシング規則を前提とするように、エイリアス分析を緩和。
- 3　const 変数と非 const 変数がエイリアスではないと仮定し、さらにエイリアス分析を緩和。

大半の状況において安全な値は 1 です。さらに最適なコード生成を許可するため、-O2 では -Xrelaxalias=2 と設定され、さらに下位レベルの最適化の場合はコンパイラによって -Xrelaxalias=1 と設定されます。

厳密なエイリアシングを有効にすると、オプティマイザでは、異なるタイプへのポインタがエイリアスしないと仮定できるようになります。以下の例においてオプティマイザでは、タイプが異なるために x と y がエイリアスしないと仮定されます。

```
void foo( int* x, long* y )
{
   *x += 2;
   *y += 3;
}
```

-O2 -Xrelaxalias=0 でコンパイル

```
foo(int*, long*):
lwz       r5,0x0(r3)         (00000000)
addic     r5,r5,0x2
stw       r5,0x0(r3)         03 (00000004) REG LSU
lwz       r3,0x0(r4)         PIPE
addic     r3,r3,0x3          01 (0000000C) REG
stw       r3,0x0(r4)         03 (00000010) REG PIPE LSU
blr
```

-O2 -Xrelaxalias=2 でコンパイル

```
foo(int*, long*):
lwz       r5,0x0(r3)         (00000000)
lwz       r6,0x0(r4)
addic     r5,r5,0x2
addic     r6,r6,0x3          PIPE
stw       r5,0x0(r3)
```

```
stw      r6,0x0(r4)        PIPE
blr
```

この仮定を無効にするコードを記述することもできます。最適なコード生成を保証するため、これは再記述される必要があります。以下の例では、厳密なエイリアシング規則が破られています。

```
long foo( short x )
{
   long y;
   short* z = ( short* ) &y; // accessing a long via a short pointer.
   z[ 0 ] = x;
   z[ 1 ] = x;
   return y;
}
```

-Xrelaxalias=2 の場合、コンパイラでは、z と y はタイプが異なるためにエイリアスしないと仮定することが許可されるため、戻り値のストアを、z へのストア前に移動し、戻り値を未定義として残すことも許可されます。relaxalias の異なる設定でコンパイルされたファイルを混ぜることは可能ですが、一般に、C99 の 厳蜜なエイリアシング規則を順守するようにソース コードを変更したほうが得策です。

このようなささいなケースでは、z と y が同じアドレスを共有していることがオプティマイザで認識可能であり、適切なコードが生成されます。ただし、これは保証されるものではなく、より複雑なケースでは、不適切なコードが生成される場合もあります。

__restrict キーワード

__restrict キーワードは、オプティマイザに追加情報を与え、より積極的にするために使用できます。これはポインタに適用できる修飾子で、これによってコンパイラでは、該当ポインタの使用によるロードとストアがその他ロードとストアとはエイリアスしないと仮定されます。__restrict キーワードの代表的な使用法は、関数の引数におけるものです。以下の例を考えてみましょう。

```
void foo( int* x, int* y )
{
   *x += 2;
   *y += 3;
}
```

他に情報がない場合、コンパイラでは x と y が同じメモリ位置をポイントする可能性があると仮定する必要があります。-O2 の場合、コンパイラでは以下のコードが生成されます。

```
foo(int*, int*):
lwz      r5,0x0(r3)        (00000000)
addic    r5,r5,0x2
stw      r5,0x0(r3)        03 (00000004) REG LSU
lwz      r3,0x0(r4)        PIPE
addic    r3,r3,0x3         01 (0000000C) REG
stw      r3,0x0(r4)        03 (00000010) REG PIPE LSU
blr
```

x へのストアは、ストアによって y の値が変更される場合に備え、y のロード前に完了しなければなりません。引数に __restrict キーワードを使用することにより、x と y が互いにエイリアスしないことをオプティマイザに保証することができます。

```
void foo( int* __restrict x, int* __restrict y ) {
   *x += 2;
   *y += 3;
}
```

以下が生成されます。

```
foo(int*, int*):
lwz       r5,0x0(r3)         (00000000)
lwz       r6,0x0(r4)
addic     r5,r5,0x2
addic     r6,r6,0x3          PIPE
stw       r5,0x0(r3)
stw       r6,0x0(r4)         PIPE
blr
```

これにより、両ロードを関数の最初に、両ストアを最後に持ってくることができるようになりました。__restrict キーワードを使用せずに、同じ情報をオプティマイザに伝えるさらに良い方法として、受信ポインタをローカル変数に渡すことが挙げられます。

```
void foo( int* x, int* y )
{
   int x_ = *x;
   int y_ = *y;
   x_ += 2;
   y_ += 3;
   *x = x_;
   *y = y_;
}
```

これにより、__restrict キーワードを使用した場合とまったく同じコードが生成されます。

```
foo(int*, int*):
lwz       r5,0x0(r3)         (00000000)
lwz       r6,0x0(r4)
addic     r5,r5,0x2
addic     r6,r6,0x3          PIPE
stw       r5,0x0(r3)
stw       r6,0x0(r4)         PIPE
blr
```

ここでも、オプティマイザではロードを関数の最初に、ストアを最後に持ってくることができます。

インライン化制御

SNC には、インライン化を制御する主要なスイッチが 3 つ用意されています。これら値への調整を行うことにより、コンパイルされるコードのサイズと実行速度に多大な影響が生じます。これは、恐らく PS3 においてユーザーが調整し得るもっとも効果的な最適化方法と言えるでしょう。たとえば、インライン化を無効にすると、SN Systems で使用されるゲーム サンプルのレンダリング時間が、12 µsecs からおよそ 80 µsecs へと変化します。この劇的な速度低下は、主に以下の理由によります。

- オブジェクト指向のコードで検出された小さな C++ メンバ関数の使用が、ゲームに共通して検出された。
- PS3 メモリのサブシステム。キャッシュ ミスやページ ミスを引き起こす関数コールでは、関数自体の命令実行において使用されるサイクルを縮小化する恐れのある、何百ものサイクル ペナルティを招きます。

PS3 におけるメリットは、かなり大きな関数のインライン化によって得られますが、これはコード サイズの増加に対してバランスをとる必要があります。Tuner を使用すると、キャッシュ ミスのためにストールを引き起こす関数を特定できます。

したがって、コードに対して最適な値を見つけるため、および Tuner からのプロファイル情報を検討するために、これらの値を十分に検証することを強くお勧めします。

これらのパラメータは、関数の最大「命令」数を表します。ここでいう「命令」とは、コンパイラの中間表現であり、その数は実際のプロセッサ命令数とは必ずしも同じではありません。-O2 以上では、特定の条件に適合する関数が自動的にインライン化されます。

インライン化は関数の大きさに依存し、これは以下に挙げる項目で制御されます。

-Xautoinlinesize

このスイッチは、ソース コード内でインラインとしてマークされることなく、コンパイラによって自動的にインライン化される関数の最大を制限します。これは、ヘッダ ファイル内の C++ メソッド定義済み内部クラスなど、非明示的なインライン関数には適用されません (-Xinlinesize を参照)。

このスイッチは -Xautoinlinesize=<n> として使用され、n には以下のいずれかが入ります。

- `0` 自動インライン化なし。
- `64` デフォルト値。
- `<n>` マークされていない関数の自動インライン化を、最高サイズ <n> 個の命令まで許可。

-Xinlinesize

このスイッチは、コンパイラによってインライン化される、非明示的なインライン関数の最大サイズを制限します。非明示的なインライン関数には、ヘッダ ファイル内の C++ メソッド定義済み内部クラスが含まれます。

このスイッチは -Xinlinesize=<n> として使用され、n には以下のいずれかが入ります。

- `0` 黙示的なインライン化なし。
- `384` デフォルト値。
- `<n>` 黙示的なインライン関数の自動インライン化を、最高サイズ <n> 個の命令まで許可。

-Xinlinemaxsize

このスイッチは、任意の 1 関数へのインライン化の最大量を制御します。このため、自動インライン化が有効な状態で、1 関数のサイズが制限されます。

このスイッチは -Xinlinemaxsize=<n> として使用され、n には以下のいずれかが入ります。

- `0` インライン化なし。
- `5000` デフォルト値。
- `<n>` 最高サイズ <n> 個の命令まで、関数へのインライン化を許可。

強制インライン化

SNC は、__attribute__((always_inline)) の使用による強制インライン化に対応しています。低レベルの数学関数などのリーフ関数 (特に -Xfastmath が使用される場合など) は、概して always_inline としてマークされる必要があります。これにより、その他のいかなるインライン化設定にも*かかわらず*、SNC は関数のインライン化を確実に行います。この場合、-O0 でも関数がインライン化されます。

例

```
#define FORCE_INLINE inline __attribute__((always_inline))

FORCE_INLINE float abs( float f )
{
    //...
}
```

6: PPU 特有の調整オプション

前のセクションでは、最適化に関するより一般的なコンパイラ オプションを取り上げました。このセクションでは、PPU のプログラミングにより関係するコンパイラ オプションについて説明します。

VMX レジスタの使用

PPU には、主に 4 つのレジスタ クラスがあります。

- 整数レジスタ
- 浮動小数点レジスタ
- VMX レジスタ
- コンディション レジスタ

これらレジスタ クラス間の変換は、非常に手間がかかり、通常ほぼ 30-70 サイクルになります。これは大抵の場合、メモリへのストア、ストア パイプライン (非常に深い) を行き来するデータの待機、その後、深いロード パイプラインを通じて再びロードされることを意味します (ロード/ヒット/ストア問題の別のインスタンス)。

これら負担を回避するため、SNC には VMX プロセッサのみを使用して多くの計算を実行する機能が備わっています。VMX では、メモリ アドレッシングと倍精度浮動小数点 (ゲームで使用されることはめったにない) を除き、PPU で実行可能な計算のほぼすべてが実行できます。

PS3 PPU ABI では、VMX ではなく、FPU を使用して浮動小数点パラメータが渡されます。ただし、コールされる関数のソース コードがコール側で使用可能な場合や、問題になっている関数に静的スコープがある場合など、適切な状況において、SNC では VMX レジスタを使用できます (事実上ローカルで ABI に違反)。これは、多数のモジュールが 1 つの (大きな) ファイルとして一緒にコンパイルされる場合、「ユニティー」ビルドのさらなるメリットとなります。この場合コンパイラでは、コールされる関数のボディを使用可能な状態にするなど、この最適化を実行するのに必要な情報が高い確率で用意されます。

VMX 変換を有効にするには、変換を中断するコール境界を回避するため、すべての関数を __attribute__((always_inline)) でマークします。

浮動小数点制御 (-Xfastmath)

-Xfastmath スイッチは、コード パフォーマンスに重大な影響を与える、多数の浮動小数点最適化を可能にします。結果コードが厳密に IEEE 標準に適合しないため、これはデフォルトでは有効ではありません。ただし、概してこれは重大な問題ではありません。

-Xfastmath=0　　保守的な設定。安全でパフォーマンスの低い浮動小数点。

-Xfastmath=1　　浮動小数点式から VMX への変換を許可し、lvlx などのベクター ロード/ストア命令の使用を許可。

分岐なし fsel 選択を有効化。

fastmath オプションでは、VMX ユニットを使用してメモリ アクセスを避けることにより、PPU PS3 アーキテクチャにおいて評判のロード/ヒット/ストア問題を回避する際に役立ちます。たとえば、

float から int、int から float への変換実行時にこれが発生します。fastmath なしでは、これらの変換によってメモリ アクセスが発生し、多くの場合ロード/ヒット/ストア ペナルティが引き起こされます。

ただし、ビデオ メモリ内に存在するデータに -Xfastmath を使用する際には注意が必要です。ビデオ メモリのベース アドレスはマクロ RSX_FB_BASE_ADDR で取得でき、現在の SDK ではベース アドレスが 0xC0000000 にありますが、これは今後の SDK 改訂で変更される可能性があります。このメモリ領域には標準メモリよりもさらに厳密なアラインメント制約があり、fastmath を含む float/int 変換で使用される、ベクター ロード/ストア命令 (lvlx など) でアラインメント例外が発生する可能性があります。これは、ビデオ メモリに存在する変数を「volatile」としてマークすることにより、パフォーマンスにおける損失はあるものの、ベクター命令が使用されなくなるため、回避することができます。この方法でのビデオ メモリのパスやアクセスは、大きなパフォーマンス ペナルティを意味するため、通常は避けるべきでしょう。

```
//
// Example: -Xfastmath
//
void max( float d[], float a[], float b[] )
{
   for( int i = 0; i != 10; ++i )
   {
      if( a[ i ] > b[ i ] )
      {
         d[ i ] = a[ i ];
      } else
      {
         d[ i ] = b[ i ];
      }
   }
}
```

```
# -O2 -Xfastmath=1 -Xassumecorrectsign=1
._Z3maxPfS_S_:
       addi   %r7,%r0,10
       addi   %r6,%r0,0
       mtctr  %r7
..L.2:
       lfsx   %f1,%r4,%r6
       lfsx   %f2,%r5,%r6
       fsubs  %f3,%f2,%f1
       fsel   %f1,%f3,%f2,%f1
       stfsx  %f1,%r3,%r6
       addic  %r6,%r6,4
       bc     16,0,..L.2
       bclr   20,0
```

```
# -O2  -Xfastmath=0 -Xassumecorrectsign=1
._Z3maxPfS_S_:
       addi   %r7,%r0,10
       addi   %r6,%r0,0
       mtctr  %r7
..L.2:
       lfsx   %f1,%r5,%r6
       lfsx   %f2,%r4,%r6
       fcmpu  0,%f2,%f1 # ~50 cycles
                        #  fcmp delay
       cror   6,0,2
       beq    1, ..L.5 # bc 12,6
       fmr    %f1,%f2
..L.5:
       stfsx  %f1,%r3,%r6
       addic  %r6,%r6,4
       bc     16,0,..L.2
       bclr   20,0
```

整数から浮動小数点への変換

整数から浮動小数点への変換は、VMX レジスタを使用して実行できます。これは、浮動小数点ユニットのみを使用した場合に比べて高速です。これは、浮動小数点から整数への変換についても同様です。

```
//
// Example: -Xfastmath
//
float result;
void count()
{
   float tot = 0.0f;
   for( int i = 0; i != 10; ++i )
   {
      float f = (float)i / 10.0f;
      tot += f;
   }
   result = tot;
}
```

```
# -O2 -Xassumecorrectsign=1 -Xfastmath=1
._Z5countv:
       addis  %r3,%r0,_Z5countv$rodata@ha
       vspltisw      %v4,0
       addi   %r4,%r0,10
       vspltisw      %v2,1
       addi   %r3,%r3,_Z5countv$rodata@l
       mtctr  %r4
       vsldoi %v5,%v4,%v4,0
       vsldoi %v3,%v5,%v5,0
       lvlx   %v6,%r0,%r3
       vspltw %v6,%v6,0
..L.2:
       vcfsx  %v7,%v3,0        # (float)i
       vadduwm       %v3,%v3,%v2    # i++
       vmaddfp       %v7,%v7,%v6,%v4
                     # (float)i / 10.0f
       vaddfp %v5,%v7,%v5      # tot
       bc     16,0,..L.2
       vspltw %v2,%v5,0
       addis  %r3,%r0,result@ha
       addi   %r4,%r0,result@l
       stvewx %v2,%r4,%r3
       bclr   20,0
```

```
# -O2 -Xassumecorrectsign=1 -Xfastmath=0
._Z5countv:
       stdu   %sp,-64(%sp)
       addis  %r3,%r0,_Z5countv$rodata@ha
       addi   %r5,%r0,10
       addi   %r4,%r3,_Z5countv$rodata@l
       addi   %r3,%r0,0
       mtctr  %r5
       lfs    %f2,0(%r4)
       lfs    %f1,4(%r4)
..L.2:
       extsw  %r4,%r3
       std    %r4,48(%sp)
       addic  %r3,%r3,1
       lfd    %f3,48(%sp)  # ~50 cycle
                           # LHS penalty
       fcfid  %f3,%f3
       frsp   %f3,%f3
       fdivs  %f3,%f3,%f1  # slow ins
       fadds  %f2,%f3,%f2
       bc     16,0,..L.2
       addis  %r3,%r0,result@ha
       stfs   %f2,result@l(%r3)
       addi   %sp,%sp,64
       bclr   20,0
```

符号拡張削除 (-Xassumecorrectsign)

-Xassumecorrectsign オプションは、32 ビットから 64 ビットへの符号拡張 (すなわち符号ビットの上位 32 ビットをコピーすること) の削除を制御します。これには、「安全」と「積極」の 2 つの設定があります。「積極」設定は、32 ビットの符号付き値または符号なし値の上位ビットが符号ビットの単純なコピーである場合に使用できます (レジスタが 64 ビット幅のため)。この仮定は、計算がオーバーフローした場合や、ポインタが 32 ビットの整数にキャストされることによって、上位 32 ビットが「意味のある」データを保持するために使用され、符号ビットの単純なコピーでなくなる場合、成立しない可能性があります。

積極設定を使用することにより、コードのサイズとパフォーマンスに重大な影響がでる場合があります。

このオプションは以下のように使用します。

-Xassumecorrectsign=0　上位 32 ビットが符号ビットのコピーであるとコンパイラが保証できるように、「安全」な符号拡張のみを削除する。

-Xassumecorrectsign=1　計算上のオーバーフローがないこと、および 64 ビット タイプ (特にポインタ タイプ) からの無チェック変換がないことを仮定する。

```
//
// Example: -Xassumecorrectsign
//
void f( int* dest, int* src )
{
   while( *src != 0 )
   {
      *dest++ = *src++;
   }
}
```

```
# -O2 -Xassumecorrectsign=1
._Z1fPiS_:
       lwz    %r5,0(%r4)
       cmpwi  0,%r5,0
       beq    0, ..L.3 # bc 12,2
..L.2:
       lwz    %r5,0(%r4)
       stw    %r5,0(%r3)
       lwz    %r5,4(%r4)
       addic  %r3,%r3,4
       cmpwi  0,%r5,0
       addic  %r4,%r4,4
       bne    0, ..L.2 # bc 4,2
..L.3:
       bclr   20,0
```

```
# -O2 -Xassumecorrectsign=0
._Z1fPiS_:
       lwz    %r5,0(%r4)
       cmpwi  0,%r5,0
       beq    0, ..L.3 # bc 12,2
..L.2:
       addic  %r5,%r4,4
       lwz    %r7,0(%r4)
       addic  %r6,%r3,4
       rldicl %r4,%r5,0,32
       stw    %r7,0(%r3)
       rldicl %r3,%r6,0,32
       lwz    %r5,0(%r4)
       cmpwi  0,%r5,0
       bne    0, ..L.2 # bc 4,2
..L.3:
       bclr   20,0
```

浮動小数点比較 (-Xuseintcomp)[4]

-Xuseintcmp オプションでは、浮動小数点比較、絶対値の取得、否定を、整数命令として書き換えることができます。PPU における浮動小数点ユニットには、非常に長いレイテンシー (> 20 サイクル) があります。また、浮動小数点比較は通常、50 サイクル以上続くパイプライン フラッシュをもたらします。このスイッチには「安全」と「積極」の 2 つの設定があります。「積極」モードでは、NAN の処理を行いませんが、負の 0 は処理されます。

-Xuseintcmp=0　安全。比較を変換しない。

-Xuseintcmp=1　積極。比較を整数処理に変換する。

```
//
// Example: -Xuseintcmp
//
void sort( float a[], unsigned n )
{
   for( unsigned i = 1; i != n; ++i )
   {
      for( unsigned j = 0; j != i; ++j )
      {
         if( a[ j ] > a[ i ] )
         {
            float tmp = a[ j ]; a[ j ] = a[ i ]; a[ i ] = tmp;
         }
      }
   }
}
```

```
# inner loop with -O2  -
Xassumecorrectsign=1 -Xuseintcmp=1
..L.5:
       lwzx   %r9,%r3,%r7
       lwzx   %r10,%r3,%r5
       srawi  %r12,%r9,31
       srawi  %r11,%r10,31
       lfsx   %f2,%r3,%r7
       rlwinm %r30,%r12,31,1,31
       lfsx   %f1,%r3,%r5
       rlwinm %r31,%r11,31,1,31
       xor    %r9,%r30,%r9
       xor    %r10,%r31,%r10
       subfc  %r9,%r12,%r9
       subfc  %r10,%r11,%r10
       cmpw   0,%r9,%r10
        # integer compare used, no delay
       ble    0, ..L.7 # bc 4,1
       stfsx  %f1,%r3,%r7
       stfsx  %f2,%r3,%r5
..L.7:
```

```
# inner loop with -O2  -
Xassumecorrectsign=1 -Xuseintcmp=0
..L.5:
       lfsx   %f2,%r3,%r7
       lfsx   %f1,%r3,%r5
       fcmpu  0,%f2,%f1
       cror   6,0,2
        # compare causes ~50 cycle delay
       beq    1, ..L.7 # bc 12,6
       stfsx  %f1,%r3,%r7
       stfsx  %f2,%r3,%r5
..L.7:
       addic  %r7,%r7,4
       bc     16,0,..L.5
```

[4] この機能はバージョン 270.1 以降でのみ使用できます。

```
addic  %r7,%r7,4
bc     16,0,..L.5
```

静的関数への変換

SNC では、外部使用に必要とされるアドレスを持たない静的関数において、特定の ABI 最適化を実行できます。コンパイル ユニット外からアクセスされない任意の関数を、「静的」とマークすることによりこれらの最適化が実行できます。

例

```
struct Vector4
{
   vector float vf_;
};

Vector4 vec;

// force this function to not be inlined for the purpose of this example
__attribute__((noinline)) Vector4 getVec()
{
   return vec;
}

vector float foo()
{
   Vector4 x = getVec();
   return x.vf_;
}
```

.. これにより、-O2 で以下が生成されます。

```
getVec():
lis      r4,0x0            (00000000)
li       r5,0x0
lvx      v2,r5,r4          03 (00000004) REG LSU
stvx     v2,0,r3           PIPE
blr

foo():
stdu     r1,-0x80(r1)
mfspr    r0,LR             02
std      r0,0x90(r1)
addic    r3,r1,0x70        PIPE
bl       0x00000028        08
li       r3,0x70           PIPE
lvx      v2,r3,r1          03 (0000002C) REG LSU
ld       r0,0x90(r1)       PIPE
mtspr    LR,r0             01 (00000034) REG
addi     r1,r1,0x80
blr
```

ただし、関数 getVec() を「静的」とマークするのみで、コンパイラでは ABI 最適化を適用でき、vecreturn 属性でタグ付けされた場合と同様に、ラッパー構造体 Vector4 をレジスタに戻すことができます。

```
foo():
stdu     r1,-0x70(r1)      (00000000)
mfspr    r0,LR             02
std      r0,0x80(r1)
bl       0x0000002C        08
ld       r0,0x80(r1)
```

```
mtspr    LR,r0             02 (00000010) REG
addi     r1,r1,0x70
blr

getVec():
```

分岐の削除 [5]

-Xbranchless スイッチでは、可能な限り、ターゲット マシンにおける比較の「分岐なし」形式が使用されます。これにより、分岐での予測が誤った場合に、パイプラインのフラッシュが回避されるだけでなく、さらなる最適化機会を目的としたループ構造の簡略化にも役立ちます。

PPU においては、整数ユニットにおける条件移動命令がないため、このスイッチは通常デフォルトでオフ (無効) となり、コールド コードの膨張が回避されます。

設定には以下の 3 つがあります。

-Xbranchless=0　無効。

-Xbranchless=1　三項演算子 (a > b ? a : b など) に対してのみ有効。

-Xbranchless=2　考えられるすべての整数比較に対して有効。

```
//
// Example: -Xbranchless
//
void max( unsigned d[], unsigned a[], unsigned b[] )
{
   for( int i = 0; i != 10; ++i )
   {
      if( a[ i ] > b[ i ] )
      {
         d[ i ] = a[ i ];
      } else
      {
         d[ i ] = b[ i ];
      }
   }
}
```

```
with -O2 -Xassumecorrectsign=1
-Xbranchless=2

._Z3maxPjS_S_:
       addi   %r7,%r0,10
       addi   %r6,%r0,0
       mtctr  %r7
..L.2:
       lwzx   %r7,%r4,%r6
       lwzx   %r8,%r5,%r6
       subfc  %r9,%r7,%r8
       subfc  %r7,%r8,%r7
       sradi  %r9,%r9,63
       and    %r7,%r9,%r7
```

```
with -O2 -Xassumecorrectsign=1
-Xbranchless=0

._Z3maxPjS_S_:
       addi   %r7,%r0,10
       addi   %r6,%r0,0
       mtctr  %r7
..L.2:
       lwzx   %r7,%r5,%r6
       lwzx   %r8,%r4,%r6
       cmplw  0,%r8,%r7
       # may mis-predict this branch
       ble    0, ..L.5
       ori    %r7,%r8,0
```

[5] この機能はバージョン 270.1 以降でのみ使用できます。

```
        addc    %r7,%r7,%r8
        stwx    %r7,%r3,%r6
        addic   %r6,%r6,4
        bc      16,0,..L.2
        bclr    20,0
```

```
..L.5:
        stwx    %r7,%r3,%r6
        addic   %r6,%r6,4
        bc      16,0,..L.2
        bclr    20,0
```

ループ展開 (unrollssa)[6]

-Xunrollssa スイッチを使用することにより、多くのループを 1 つの基本的なブロック ループに変換させることができます。このオプションでは、該当命令内のループの最終サイズを示すパラメータを利用します。ただし、通常はさらなる最適化が行われるため、最終的な命令数が大幅に少なくなることから、ここでは大きな数値を指定する価値があります。

-Xunrollssa=0　　ループを展開しない。

-Xunrollssa=10　非常に小さなループのみを展開する。

-Xunrollssa=30　より大きなループを展開する。

-Xunrollssa=100　最大規模のループを展開する。実際のコードで役に立つ可能性は低いものの、ベンチマークやその他小規模なサンプルで役立つ場合もある。

設定を大きくした場合、フェッチされる各コールド命令では約 20 サイクルが必要となるため、コールド コードの実行速度が遅くなります。

```
//
// Example: -Xunrollssa
//
// variable iteration loop unrolling
//
void f( int* dest, int* src )
{
   while( *src != 0 )
   {
      *dest++ = *src++;
   }
}
```

```
# with -O2 -Xunrollssa=100 -
Xassumecorrectsign=1
._Z1fPiS_:
        lwz     %r5,0(%r4)
        cmpwi   0,%r5,0
        beq     0, ..L.4 # bc 12,2
..L.2:
        lwz     %r5,0(%r4)
        stw     %r5,0(%r3)
        lwz     %r5,4(%r4)
        cmpwi   0,%r5,0
        beq     0, ..L.4 # bc 12,2
          # Branch suffers from a 4 cycle
```

```
# with -O2 -Xunrollssa=0
._Z1fPiS_:
        lwz     %r5,0(%r4)
        cmpwi   0,%r5,0
        beq     0, ..L.3 # bc 12,2
..L.2:
        lwz     %r5,0(%r4)
        stw     %r5,0(%r3)
        lwz     %r5,4(%r4)
        addic   %r3,%r3,4
        cmpwi   0,%r5,0
        addic   %r4,%r4,4
```

[6] この機能はバージョン 270.1 以降でのみ使用できます。

```
        # branch after branch delay
       stw    %r5,4(%r3)
       lwz    %r5,8(%r4)
       addic  %r3,%r3,8
       cmpwi  0,%r5,0
       addic  %r4,%r4,8
       bne    0, ..L.2 # bc 4,2 <-
        # This branch doesn't
..L.4:
       bclr   20,0
```

```
       bne    0, ..L.2 # bc 4,2
         # Branch suffers from a 4 cycle
         # branch-after branch delay
..L.3:
       bclr   20,0
```

```
//
// Example: -Xunrollssa
//
// constant iteration loop unrolling
//
void add( float d[], const float a[], const float b[] )
{
   for( int i = 0; i != 3; ++i )
   {
      d[ i ] = a[ i ] + b[ i ];
   }
}
```

```
# with -O2 -Xunrollssa=100 -
Xassumecorrectsign=1
._Z3addPfPKfS1_:
       addi   %r6,%r0,0
       addc   %r7,%r4,%r6
       lfs    %f1,0(%r4)
       lfs    %f2,0(%r5)
       addc   %r4,%r5,%r6
       fadds  %f1,%f1,%f2
       addc   %r5,%r3,%r6
       stfs   %f1,0(%r3)
       lfs    %f1,4(%r7)
       lfs    %f2,4(%r4)
       fadds  %f1,%f1,%f2
       stfs   %f1,4(%r5)
       lfs    %f1,8(%r7)
       lfs    %f2,8(%r4)
       fadds  %f1,%f1,%f2
       stfs   %f1,8(%r5)
       bclr   20,0
```

```
# with -O2 -Xunrollssa=0 -
Xassumecorrectsign=1
._Z3addPfPKfS1_:
       addi   %r7,%r0,3
       addi   %r6,%r0,0
       mtctr  %r7
..L.2:
       lfsx   %f1,%r4,%r6
       lfsx   %f2,%r5,%r6
       fadds  %f1,%f1,%f2
       stfsx  %f1,%r3,%r6
       addic  %r6,%r6,4
       bc     16,0,..L.2
       bclr   20,0
```

7: PPU のコーディング形式

このセクションでは、PPU のプログラミングに関する全般的な点についていくつか説明し、PPU に関する主要ポイントを示すコード例も取り上げます。

レジスタ クラス間での転送

メイン データ レジスタ クラス間での変換 (スカラー、浮動小数点、ベクター) では、変換の際にメモリ アクセスが行われるため、犠牲が大きくなります。以下に示すコードは、単純な例です。

```
//
// Example: the three main register classes of the PPU
//
// Unoptimized, each of these copies causes a long and extremely inefficient sequence.
//
volatile int i;
volatile float f;
volatile vector float vf;
void shuffle()
{
   i = (int)f;
   i = vec_extract( (vector signed int)vf, 0 );
   f = (float)i;
   f = vec_extract( vf, 0 );
   vf = (vector float)( i );
   vf = (vector float)( f );
}
```

ただし特定の状況において、VMX レジスタが以下のように使用される場合は、float/int 変換を効率的に処理できます。

```
//
// Example: SNC's automatic vmx conversion
//
// it is easy to convert from int or float to vmx when the whole expression can be
// done in VMX registers. VMX can execute both integer and float instructions,
// so mixed int and float code can be handled.
//
int i;
float f;
vector float vf;
double d;
void good()
{
   // this expression is done using the vmx
   i = (int)f;

   // these expressions are also done using the vmx
   i = (int)( f + 1.0f );
   i = (int)f + 1;
   // conversions to vmx are done by re-writing the incomming expressions.
   // in this case, f can be loaded as a vmx register and the expression
   // can be calculated using vmx.
   vf = (vector float){ f, f + 1, f + 2, 0 };
}
```

```
float bad( float float_param )
{
   // this expression is very inefficient. double is not supported on the vmx.
   i = (int)d;
   // this expression is difficult to do reliably on the vmx
   // as accidental NANs can cause failures
   f = sqrtf( f );
   // parameters cannot be converted to vmx without a large penalty.
   i = (int)float_param;
   // outgoing float values cannot be extracted from VMX without a penalty.
   return (float)i;
}
```

インライン化された定数の使用

以下は、コストの大きなメモリ アクセスを回避するために使用できる別の方法です。

```
//
// Example: SNC inline constants are much more efficent than loading from memory.
//
// SNC "prewarms" constants by having them located at the start of a function.
// Loading a constant from memory often needs a 500+ cycle L2 cache miss or ERAT
// refill.

vector float a, b;
vector unsigned int c, d;

void good()
{
   // this constant is loaded from prewarmed memory.
   a = (vector float){ 0xaabbaabb, 0xbbcdefaa, 0xddbba0bb, 0x0bbcdeaf };

   // this constant only needs 4 bytes.
   a = (vector float)( 1.0f );

   // this constant may be generated by shifts. ( -1 << -1 )
   b = (vector float)(vector unsigned int)( 0x80000000 );

   // these constants can be generated with single vmx instructions
   c = (vector unsigned int)( 1 );
   d = (vector unsigned int){ 0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f };
}
```

ベクター キャストに対する共用体の使用

多くのサンプル コードでは、タイプ間のキャストに共用体が使用されます。キャストがベクター タイプに関連する場合は、共用体を利用するよりも、明確なキャストを使用する方が効率的です。その場合、コンパイラはパターンを認識でき、ベクターユニットを使用することができます。ただしその場合は、fastmath オプションを有効にする必要があります。

```
//
// Example: there is no need to use unions for vector casts.
//
// SNC can "untangle" casts using unions, but the compiler will have less work to
// do if you use a cast directly. This will probably generate better code.
//
// Try to avoid using unions where possible in high performance code.
```

```
vector float vf;
__attribute__( ( always_inline ) )
inline void good( float x, float y, float z, float w )
{
   vf = (vector float){ x, y, z, w };
}
```

ベクター比較のレイテンシー (vec_all_* と vec_any_*)

ベクター ユニットには深いパイプラインが存在するため、ベクター処理 (Altivec 組み込み関数など) の結果が出るまでには長いレイテンシーがあります。これはたとえば、テストによって終了条件に影響がでるループなどにおいて、問題となりえます。可能な場合は、該当条件の使用を避けるために、コードの変更が必要となります。

```
//
// Example: vector compare
//
// The altivec vec_all_* and vec_any_* intrinsics require a huge latency before
// the result can be used for a branch.
// As a result, they are really too slow to use.
// So try to avoid loops that "early out" on vector compares.


// very common style of frustum plane visibility test.
bool badVisibilityTest( vector float frustumPlanes[], vector float posAndRadius )
{
   for( int i = 0; i != 6; ++i )
   {
      vector float dot = vec_madd( frustumPlanes[ i ], posAndRadius, (vector
      float)0.0f );
      dot = vec_add( dot, vec_sld( dot, dot, 8 ) );
      dot = vec_add( dot, vec_sld( dot, dot, 4 ) );

      // this may take 100+ cycles per loop = 600 cycles in total
      if( vec_any_ge( dot, (vector float)0.0f ) )
      {
         return false;
      }
   }
   return true;
}

// smarter test.
void betterVisibilityTest( vector float frustumPlanes[], vector float posAndRadius,
unsigned char* result )
{
   vector float pass = (vector float)(-1.0f);
   for( int i = 0; i != 6; ++i )
   {
      vector float dot = vec_madd( frustumPlanes[ i ], posAndRadius,
                                                    (vector float)(0.0f) );
      dot = vec_add( dot, vec_sld( dot, dot, 8 ) );
      dot = vec_add( dot, vec_sld( dot, dot, 4 ) );
      pass = vec_and( pass, dot ); // and all the sign bits
   }

   // Note: do not read this result immediately as you will cause a LHS stall.
   *result = vec_extract( (vector unsigned char)vec_cmpge( pass, (vector float)0 ),
    0);
}
```

参照渡しではなく値渡しを使用

参照 (アドレス) によって渡されるパラメータは、ロード/ヒット/ストア ペナルティの原因となる傾向があります。サイズが大きいデータを値渡しすることは効率的ではありませんが、スカラーやベクタータイプの場合は常に有益となります。

```
//
// example: pass and return by value when possible
//
//
// parameters passed by reference cause LHS stalls
// because the value written to the structure in the caller
// is immediately read back by the callee
//
// Similarly, results passed by reference will encounter a similar problem
// if they are immediately re-read by the caller.
//
// The Darwin 64 abi has a more efficent way of passing by value and should
// be used wherever possible. With the d64 abi, values are passed in their
// correct register classes.

struct SimpleRendererCommand
{
   unsigned char function;
   float x, y, z;
};

void draw( SimpleRendererCommand &cmd );

__attribute__( ( noinline ) )
void badExecute( SimpleRendererCommand &cmd )
{
   // LHS here..
   switch( cmd.function )
   {
      case 1:
      {
         draw( cmd );
      } break;
   }
}

void badCall()
{
   SimpleRendererCommand cmd;
   cmd.function = 1;
   badExecute( cmd );
}

struct MyBetterRendererCommand
{
   unsigned char function;
   float x, y, z;
} __attribute__( ( d64_abi ) );

void draw( MyBetterRendererCommand& cmd, float x, float y, float z );

// if this is too big to inline. It won't be a disaster.
__attribute__( ( noinline ) )
void betterExecute( MyBetterRendererCommand cmd, MyBetterRendererCommand
&polymorphicCmd )
{
   // now no LHS here..
```

```
   switch( cmd.function )
   {
      case 1:
      {
         // functions that absolutely need pointers can use the pointer argument
         draw( polymorphicCmd, cmd.x, cmd.y, cmd.z );
      } break;
   }
}

void betterCall()
{
   MyBetterRendererCommand cmd;
   cmd.function = 1;
   betterExecute( cmd, cmd );
}
```

8: SN リンカーの使用

FSELF (Fake-Signed ELF)

SN リンカーには、FSELF (Fake-Signed ELF) フォーマットに出力する機能が最初から搭載されています。FSELF フォーマットは、PS3 Reference Tool と Debugging Station における直接使用に適しており、make_fself プログラム (PS3 SDK に付属) を実行する必要がなくなります。また、SN リンカーの内部 FSELF 出力は make_fself プログラムよりも大幅に高速であるため、プロジェクトのビルド時間が短縮されます。

FSELF 出力は、--oformat=fself コマンドライン スイッチをパスすることによって有効化されます。なお、これが SN コンパイラ ドライバを使用してコールされる場合、このオプションは -Wl,--oformat=fself となります。

SN リンカーには多数の重要な一般的および PS3 特有の最適化が搭載されています。これらはゲーム パフォーマンスに大きく影響しますが、ゲーム サイズにはさらに重大な影響を与えます。コンパイラ オプションと同様に、このセクションでは、一般的なオプションと (PS3) アーキテクチャ特有の最適化について取り上げます。

SN リンカーでのデッド ストリップ

はじめに

最近のプロジェクトの規模を考えると、未使用のコードやデータをプログラマが手作業で取り除くことは難しいと言えます。また、C/C++ では、最終的な実行ファイルで使用されない定義が頻繁にコンパイラで発せられることになります。SN リンカーには、使用されていない (デッド) コードやデータを、リンク フェーズの間にプログラムから削除する機能が搭載されており、最終的な出力ファイル サイズを削減できます。

デッドストリッピングの基本的なアプローチは、プログラムの静的参照グラフを作成し、それを分析することにより、未使用または重複しているエレメントを特定することです。この分析は、リンカーのパフォーマンスにさまざまなレベルで影響を与えます。こうしたことから、デッド ストリッピング有効化の選択は、プロジェクト サイズ (非デバッグ ビルド用に用意されたもの) に依存することになります。ただし、多くの開発者が全ビルドに対してデッド ストリッピングを有効にすることにより、節約されるメモリを考えると、このトレードオフは十分価値があるものと言えます。

以下に説明するすべての SN リンカー コマンドライン オプションは、GCC ツールチェーンと同じように、SN コンパイラ ドライバを使用してリンカーを呼び出す際、-Wl の後に続けます。VSI (Visual Studio 経由) を使用する場合、コマンドは、ProDG VSI プロジェクト プロパティのリンカーへの「追加オプション」として追加します。

デッドコードのストリッピング

デッドコードのストリッピング モードでは、実行ファイルのコードが分析され、使用されていない関数が特定されます。さらに、TOC セクションの分析も行われ、GCC でビルドされたライブラリを静的にリンクすることによって取り入れられる TOC データを削減できます。これは、デッドストリッピング オ

プションの中でも最速のオプションです。概算として、このプロセスによってリンク時間が約 50% 長くなります。

デッドコード ストリッピングは、プロジェクト プロパティでリンカーへの「追加オプション」として (VSI の使用時)、またはコマンドラインで --strip-unused を使用することによって有効化できます。

デッドコードとデッドデータのストリッピング

デッドコードやデッドデータのストリッピング モードでは、リンカーにより、実行ファイルのコードとデータ セクションが分析され、使用されていないコードとグローバル データの両方が削除されます。また、コードとデータの相互関係も分析されるため、デッドコードが他の未使用データを参照する場合、デッドデータ ストリッピングも発生します (逆もまた同様)。このモードでは、デッドコード ストリッピング単独の域を超えるオーバヘッドがあまり増加しないため、ビルド時間の増加とサイズ削減との間における優れた妥協策となります。

デッドコードとデッドデータのストリッピングは、--strip-unused-data コマンドライン スイッチを使用、または VSI でのリンカーに対する「追加オプション」によって有効にできます。

コードとデータの重複削除

このモードでは、重複オブジェクト (同一シーケンスの命令から成る関数や、同じ値を持つ読み込み専用データなど) の検出と削除が行われます。重複削除では、デッドストリッピング単独の場合よりもさらに出力ファイル サイズが削減されるものの、これには局所性を削減する傾向があるため、キャッシュ ヒット率に悪影響を与える場合があります。また、重複削除はデッドストリッピング オプションの中でもっとも時間がかかります。

重複削除は、--strip-duplicates コマンドライン スイッチと、--strip-unused または --strip-unused-data を合わせて指定することによって有効になります。--strip-unused または --strip-unused-data の選択では、デッドストリッピング モードに加え、重複削除の範囲も決定します。VSI を使用している場合、これらのオプションは ProDG VSI プロジェクト プロパティ メニューで指定できます。

レポート機能

リンカーでは、デッドストリッピング オプションの使用による「節約」を説明するレポートを生成できます。デッドストリッピングを行わない状態でリンクすると、このレポートには、節約サイズの予測が表示されます。それ以外の場合、レポートには、ストリップされたまたは重複削除されたオブジェクトに加え、対応するサイズ節約情報が表示されます。またこのレポートには、ストリップされていないオブジェクトとその理由も表示されます。これは、オブジェクトがストリップされなかった理由を確認する場合や、関数やオブジェクトの使用状況を確認する場合に便利です。詳細は、SN リンカーのマニュアルを参照してください。

デッドストリッピングのレポートは、--strip-report=filename コマンドライン スイッチを使用することにより、有効にできます。

デッド ストリッピングとデバッグ データ

デッドストリッピングではほとんどの場合、実行ファイルとデバッグ データとの間のやり取りが保存されます。ただし、時々若干の問題が発生することがあります。このような問題は、コードに対して実行された変換、およびデバッグ データでこれらを表現することの難しさに起因します。複数のソース関数が同じオブジェクト コードにマップされるため、重複削除にはこの傾向があり、場合によってはプログラマやデバッギング ツールにとっての問題を引き起こすこともあります。

9: PPU 特有の最適化

「TOC 復元なし」モード

バックグラウンド

PS3 PPU ABI では、C/C++ プログラムにおけるグローバル データへのアクセスに TOC を使用するよう指示します。TOC 自体は、グローバル データ アイテムに対応する 32 ビット アドレスのリストです。ただし、TOC へのアクセス メソッドではそのサイズが 64 KB に制限されるため、1 プログラム内に複数の TOC (「*TOC 領域*」としても知られる) が必要となる場合もあります。TOC ポインタは汎用レジスタの使用によって利用可能となるため、PS3 PPU ABI では、この *TOC レジスタ*が関数コールの前に正しく設定され、その後復元される必要があります。ABI では、関数コールの後に *nop* 命令が続く必要があり、これによってリンカーでは TOC レジスタを復元するためのコードにパッチが行えます。また、関数ポインタや仮想関数を使用したコールでは、TOC レジスタを正しく設定するために、適切な処理が必要です。

SNC のアプローチ

SNC では、グローバル データへのアクセスに TOC を使用しません。代わりにリンカーと連携し、必要なデータ アドレスが生成される 2 つの命令が連続で発行されます。この方法では、コード サイズがわずかに増加するものの、TOC セクションのサイズが削減され、TOC を使用したデータ アクセスに必要とされる余分な命令も除去できるため、コードではメリットを得ることができます。この結果、キャッシュ使用量の改善や、グローバル データ アクセスにおける平均レイテンシーの低減も実現します。

ただし、ABI では他コード (PRX ライブラリ内に存在する GCC ビルド コードなど) との相互運用のために、関数コール後の nop 命令が必要となります。しかし大抵の場合、コードの大半がソースからビルドされるため、この要件はゲーム コードに対しては過剰と言えます。これは、結果的に不必要な *nop* 命令を大量に生み出すこととなり、実際のオブジェクト コードの検証した結果、典型的なゲームでは 1,000 から 100,000 の *nop* 命令が含まれることがわかっています。これは、不必要な *TOC 復元*プレースホルダー命令によって最高数百キロバイトのコード サイズが占有されることを示しています。

なお、コンパイラでは、アーキテクチャ面やパフォーマンス面での理由に対してコードをアラインするため、ABI に関するこの使用法とは無関係に、その他に *nop* 命令が生成されることもあります。

「TOC 復元なし」モード

「TOC 復元なし」モードは、SNC と SN リンカーの併用時に利用できます。これにより、TOC-復元プレースホルダー命令の省略が可能となり、関数ポインタや仮想関数を使用したコールに必要な命令シーケンスが削減され、結果としてコードをより小さくまとめることができます。大半のコードでは「TOC 復元なし」モードにより、悪影響を受けることなくコード サイズを削減できます。

「TOC 復元なし」モードは、以下の方法でコンパイラ ドライバを呼び出すことによって有効にできます。

```
ps3ppusnc -Xnotocrestore=2 -Wl,--notocrestore <other options, filenames, etc.>
```

また、コンパイルとリンクを別々に行う場合は、以下のコマンドで行います。

```
ps3ppusnc -c -Xnotocrestore=2 <other options, filenames, etc.>
ps3ppuld --notocrestore <other options, filenames, etc.>
```

このモードにおいてコンパイラでは、すべての関数コールのターゲットが同じ TOC 領域を共有すると仮定され、TOC レジスタの保存や復元を利用するすべてのコードは出力されません。同時にリンカーでは、最終的な実行ファイルに 1 つの TOC 領域のみが存在することが強制されます。静的ライブラリなどの影響により、複数の TOC 領域が必要であるとリンカーで判断された場合は、エラーが発行され、リンク フェーズが失敗となります。この場合は、TOC データ量の削減、または「TOC 復元なし」モードの無効化が選択肢となります。

PRX ライブラリの呼び出しは、リンカーが挿入するスタブ関数経由で行われるため、たとえ PRX ライブラリの呼び出しがあったとしても、1 つの TOC 領域のみが存在するという仮定は成り立ちます。「TOC 復元なし」モードではリンカーによってスタブ関数が書き直され、TOC ポインタが必要に応じて修正および復元されます。PRX ライブラリへのコールを非常に多く行うコードでは、「TOC 復元なし」モードにおいてコード サイズの縮小があまり見られない場合もあります。これは、置換される PRX スタブ関数がオリジナルのものよりも若干大きいためです。詳細は、SN リンカー マニュアルを参照してください。

使用制限

原則として、PS3 PPU ABI では「TOC 復元なし」モードの安全性が保証されています。ただし、多くの最適化と同様に、実際のコードで「TOC 復元なし」モードが正しく処理されない例外もあります。これらのケースは、幸いまれで回避も容易に行えます。

これらの例外ケースは、C/C++ において、PRX ライブラリや別の TOC 領域を持つ関数の呼び出しを、リンカーが認識すること無しに行えることに起因しています。これが起こり得る 2 つのケースとして、関数ポインタを経由した関数呼び出しと仮想関数呼び出しが挙げられます。「TOC 復元なし」モードでコンパイルされたコード内でこのような呼び出しを行った場合、TOC 領域に正しくアクセスできません。ただし、この方法での PRX コードから「TOC 復元なし」コードへのコールは安全に行えます。

間接的な関数呼び出しを行う場合、その呼び先が確定していないため、ビルド時にこの問題を発見することはできません。これらケースに関する詳細は、SN リンカー マニュアルに掲載されているので、必要に応じて参照してください。さらに重要な点として、SNC ツールチェーンの「TOC 復元なし」モードでビルドされた PRX ライブラリでは、この制限から除外されることが挙げられます。ただし、すべてのシステム PRX ライブラリは現在 GCC ツール
チェーンでビルドされています。

要約すると、「TOC 復元なし」モードの使用時には、PRX ライブラリの関数へのポインタは取らないこと、そして PRX ライブラリで定義された仮想関数を含む C++ クラスを使用しないこと、となります。

PRX ライブラリ コードの回避策

上記のケースのいずれかにおいて、PRX ライブラリへのコールが避けられない場合でも、「TOC 復元なし」モードを使用することにより、コード ベースの大半を改善することが可能です。

SNC には、各関数ごとに「TOC 復元なし」モードを制御できる、#pragma 指令が用意されています。これらの指令を使用すると、対象となる関数は、「TOC 復元なし」モードが無効化されている場合と同様に動作します。以下は、この特性を実証する例です。

```
#pragma control %push notocrestore=0
#pragma noinline
void invoke_callback (void (*callback) ())
{
   callback (); // Target may safely be in a different TOC region.
}
#pragma control %pop notocrestore
```

この #pragma noinline を使用した例では、「TOC復元なし」モードの無効化を否定する関数 invoke_callback() をコンパイラでインライン化しません。[7]

[7] 270.1 以前のコンパイラ使用の場合は、Cell GCM ライブラリが ”no TOC restore” モードでは安全ではないいくつかのコードを含むことにご注意下さい。このライブラリを使う場合、CELL_GCM_SNC_NOTOCRESTORE_2 マクロがコンパイル中に定義されている必要があります。このマクロ定義は危険なコードを安全なそれへと置換します。SNC 270.1 ではこの必要はありません。