SN SYSTEMS
Sony Computer Entertainment Group

White Paper: SN008

How to get the best from SNC PPU toolchain for PLAYSTATION®3 development

SN Systems Limited
Version 1.0
February 4, 2009

Contents

1: Introduction

This document provides an overview on how to maximize the potential of the SNC PPU toolchain, mainly the compiler (SNC) and linker, in developing games for the Sony PLAYSTATION®3 (PS3). This document describes hints and tips on using tools for the PS3 and coding for the PS3 PPU processor.

We describe features and use examples from SNC 270.1 but unless explicitly noted, the examples will work on earlier versions of SNC too. However, the code generated will be different to the SNC 270.1 based examples here.

As well as a compiler and linker, the SNC PPU toolchain includes an assembler/disassembler and other binary tools. The linker and other tools that make up the SNC PPU toolchain are all based on SN Systems' proprietary technology that has been developed and deployed on other game platforms.

SNC is an ISO compliant C++ and C compiler for the PS3 PPU processor. The compiler is derived and developed from the Apogee compiler technology which has been developed over the last 20 years. The Apogee compiler technology is a mature technology which has been ported to many targets such as MIPS, Arm/Thumb as well as PowerPC. The compiler uses the industry standard EDG (Edison Design Group) frontend for C++ parsing and semantic analysis.

The following aspects of the compiler have been enhanced and developed:

- The PowerPC code generator has been optimized for C++ programming game development. For example, the inlining of class member functions is crucial for performance in game development, but too much inlining results in bloated code which must be avoided. The inlining strategy of the compiler has been developed with a strong focus on striking a balance between these two goals.
- The vector math capabilities of the PowerPC are specifically targeted.
- Typical PS3 game projects are very large software projects in terms of the numbers of source files and lines of code. The compiler has been structured to make it very suitable for handling such large software projects.

Optimizers of the compiler include the standard compiler optimizations such as global and local register allocation and instruction scheduling, but also includes a set of optimizations based on the modern SSA (Static Single Assignment) format which include a set of PowerPC specific optimizations. The optimizers are developed and tuned based on real game code performance rather than targeting any specific code benchmarks. A balance between code size and performance is always maintained because this is very important on the PS3 platform.

A key feature of the toolchain is that both compiler and linker object files fully conform to the PPU ABI and therefore are freely interchangeable with GCC originated object files.

The other major component of the SN toolchain is the linker. Similar to the compiler, the linker has been developed and optimized with real game codes and is therefore tuned to games development for the PS3. It has options for reducing code size and size of debug data by stripping out unused or duplicated sections to fit within the space restrictions of the PS3 and optimizes many common sources of bloated code due to C++ idioms such as templates.

The rest of this document explains the steps to build projects by using the SN toolchain. These steps are:

- Setting up the build environment, optimizing for build speed, using distributed build farms and pre-compiled header files.
- Measuring performance and size to find the cause of performance issues.
- Performance tuning for SNC. This concentrates mainly on performance but also some size issues are described. We look at both generic compiler options and PPU specific options.
- Specific coding issues on the PS3 PPU Architecture. These are mostly toolchain independent and reflect the architecture of the PS3 PPU.
- Performance tuning with the linker. This is mainly in the area of code size.

References

The following SNC PPU Toolchain documents are in the help folder of the toolchain distribution.

ProDG_PS3_Compiler: ProDG Compiler documentation

ProDG_PS3_Linker: ProDG Linker Documentation

ProDG_PS3_Utilities: Documentation on the build utilities, ps3name.exe, ps3snarl.exe & ps3bin.exe

2: Build environment

This section considers some of the factors that influence project build speeds.

Host machine considerations

Host memory

The more physical memory a machine has, the better it performs. The SNC PPU toolchain is able to cope with large physical and virtual memory addresses. This enables the tools to address up to 4 GB for the 64-bit Windows platform and 3 GB for the 32-bit Windows platform by using the /3gb boot option. Information regarding Microsoft's boot configuration data can be found on the MSDN Web site[1].

The linking process is almost entirely I/O bound. Insufficient RAM results in an increase to the use of the virtual memory swap file and can drastically affect the link. It is recommended that the host machine has at least half as much memory again as the total size of the object and library files that are included in the link. Two GB is usually considered reasonable for a full link of PS3 games[2].

Host processors

Although SNC is a single threaded process, larger projects can take advantage by building multiple sources in parallel. The number of parallel builds on a local machine can be easily configured for build systems such as VSI, SN-DBS. However, the use of parallel builds may not always be beneficial due to memory contention. The host machine should have enough memory to allow parallel compilations without swapping. It is usually better to serialize builds rather than have parallel builds which cause swapping.

Visual Studio and VSI

The SN toolchain can be driven from Visual Studio by using the SN Visual Studio Integration (VSI). If you are porting an existing GCC based project, the VSI tool addsnconfig.exe can be used to add SNC to the existing project.

It is recommended to reduce the usage of static archives[3] wherever possible to improve build times. VSI can only perform individual links, which could also cause stalls in parallel/distributed builds that are waiting on an archive link before proceeding with builds of the next project. Excessive use of static archives can also greatly affect the link time performance because the libraries must be repeatedly scanned by the linker to ensure that the correct object files are incorporated into the link. Static archives are sometimes only used as a list of files and the work carried out by the archive utility is redundant.

[1] http://msdn2.microsoft.com/en-us/library/ms791558.aspx
[2] SN-Linker manual
[3] Statically linked libraries

SN-DBS

SN-DBS allows concurrent compilations to be performed on multiple machines and on multi-core machines.

For best build performance the following points should be taken into account:

- Reduce project dependencies to allow projects to be built in parallel.
- Archive linking times cause stalls because these represent (often unnecessary) synchronization points.

It is important to note that PCH (Precompiled Header)-based projects work with SN-DBS, but the PCH is not distributed and is therefore not used. However, SN-DBS is typically faster than local builds for PCH-based projects.

Additional optimization strategies for SN-DBS can be found in the SN-DBS manual.

Precompiled header files

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that include them are relatively small. SNC provides a mechanism for taking a snapshot of the compilation state at a particular point and writing it to a disk file before completing the compilation. When recompiling the same source file or compiling another file with the same set of header files, it can recognize the "snapshot point", verify that the corresponding PCH file is reusable, and then read it back in. With the correct circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files are large and the I/O traffic can be such that the benefit of PCH is lost.

The relative overhead incurred in writing out and reading back a PCH file is quite small for reasonably large header files. In general, it does not cost much to write a PCH file out if it is not used, but if it is used, it invariably speeds up compilation. The problem is that the PCH files can be quite large, from a minimum of about 250 KB to several megabytes or more.

Therefore, despite the faster re-compilations, PCH processing is not likely to be justified for an arbitrary set of files with non-uniform initial sequences of pre-processing directives. Rather, *the greatest benefit occurs when a number of source files can share the same PCH file*. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large PCH files can be minimized without giving up the advantage of a significant speed increase in compilation times. Projects that are consistent when including header files will benefit from PCH. For example, if all source files use a single header file which itself includes all the main header files in a project, you will see a significant reduction in the compile time. By contrast, for a project that includes its header files in an ad-hoc fashion, you will see no benefit and possibly an increase in the compile time. PCH files do not always make a positive impact.

3: Measuring performance

Before you optimize game performance, it is vital to measure performance rather than assuming the location of performance bottlenecks.

The PS3 Tuner can provide detailed information on overall game performance. Tuning for the PS3 platform involves many design tradeoffs, for example SPU processing, but here we concentrate on PPU performance. With Tuner, you can perform hot-spot analysis by using PC sampling.

Consult the Tuner documentation to set up a Tuner project and capture PC samples for your project. After capturing the data, ignore the per-frame trace in the bottom pane and pull down the divider to analyze the global data which provides more detail.

The left pane displays a list of the most sampled functions, but not necessarily the biggest hot spots in the game because the PC samples are averaged over the whole of the function and each function is typically many hundreds of instructions. Mark the frequently executed functions with __attribute__((hot)). This will guide the compiler in its inlining and code placement decisions.

```
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
```

In this example, for 35% of the time the processor is idle, 14% is spent in the OS (mostly “sleep” calls in the audio code) and 5% is utilized by various other OS processes and functions.

Therefore, about 46% of the CPU remains to run the game.

Much of this is spent in the first few instructions of the function DoIt::DoIt[4]. About 10% of this is virtual function call overhead before entering the function. In this game, virtual function overhead accounts for about 5% of the frame time.

[4] Note the names of the functions in this library have been made anonymous. This style of function naming is not encouraged!

In this particular graphics library, perhaps less than 5% of the time is spent performing calculations and the rest is spent moving memory and calling functions. This is typical for game engines ported from PC environments. However, the physics library is far more effective.

If you double-click the hottest function, you obtain the following trace. Note the values in the left column – these are the PC sample hit counts. A value of 500 or more represents a bad delay that should be fixed. These values are marked in parentheses.

```
          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        xxxxxxxxxxx
  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
```

To determine the origin of the hotspots, find the instructions that define the registers used by each hotspot. The stall is never displayed on the instruction itself, but always derives from one of the inputs to the hotspot instruction being delayed.

It is recommended to use `-Os` wherever possible for optimal size and speed for the PS3. Hot functions can be selectively compiled at higher levels of optimization.

4: Quick guide to optimization settings

This section is a quick guide to generally useful optimization settings for build projects. A release build aiming for highest performance should normally be built with:

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

Note that `-O` is equivalent to `-O2`. If code size is a significant factor, you can use this option:

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

Note that `-Os` is essentially `-O2` with reduced inlining.

A typical debug build should use the `-Od` option as follows. This generates optimized but debuggable code:

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

For release builds, the following settings are generally recommended for the linker:

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

For debug versions of these builds, it is only necessary to add the `-g` flag.

5: Generic performance tuning with SNC

This section covers more generic (i.e. non PS3 specific) optimizations for SNC.

General optimization options

The main optimizer in SNC is controlled by the command line switch `-O<n>` (where `<n>` is from `0` to `3`).

- With `-O0` the optimizer is not run. (This is the default with no –O option specified)
- With increasing values of `n`, more optimizations are applied as detailed below. Note that the current compiler does not add more optimizations after –O2, i.e. –O3 is the same as –O2. However as the compiler is developed more of the optional optimizations will be made available via –O3.
- With `–Os` the compiler is instructed to optimize for space, possibly at the expense of speed.
- With `–Od` the compiler is instructed to avoid optimizations, such as instruction scheduling, that make code more difficult to debug. This mode is designed to generate code with reasonable runtime performance that is still easy to debug.

It is recommended to use `-Os` wherever possible for optimal size and speed for the PS3. Hot functions can be selectively compiled at higher levels of optimization.

`-O0`	No optimization (the default), no inlining (except forced inlining).
`-O1`	No optimization, inlining allowed.
`-O2`	Full optimization with reasonable compile time performance.
`-O3`	Full optimization plus more time-consuming optimizations.
`-Od`	Debuggable optimized code (should be used with `-g`).
`-Os`	Optimized code avoiding optimizations that may increase code size.

Alias analysis

Alias analysis is important on the PS3 because it allows the compiler to avoid costly memory accesses by knowing the side effects of updating through pointers and array accesses. The SNC optimizer must consider the results of its alias analysis phase when judging whether or not to make optimizations. This often leads to the optimizer to be more conservative due to potential aliasing issues. The C99 language standard defines a set of alias rules based on types and this is referred to as *Strict Aliasing*. Generally, programs which conform to C99 rules do not have any problems with the alias analysis of the compiler. Note the same alias rules apply to C90 and C++03 as well as the C99 standard.

Two pointers are said to alias one another when they point to the same object or address. If the alias analysis phase is able to determine that two pointers definitely do not alias each other then it allows the optimizer to make a number of more aggressive optimizations and the scheduler to perform much more aggressive scheduling. If the alias analysis phase is unable to determine whether two pointers alias each other then it must make the assumption that they do, and hence not allow the optimizer or scheduler to make aggressive optimizations.

For this reason, it is important to write code in a manner that provides the optimizer as much information as is possible. This can be achieved by using specific coding styles in addition to keywords that may provide hints (such as __restrict). It is also possible to allow the compiler to make certain assumptions about the code (specifically that it adheres to the strict aliasing rules). However, the code must then never break these assumptions otherwise incorrect code generation may occur.

Strict aliasing

Strict Aliasing is essentially assuming C99 standard rules. By making this assumption, the compiler is able to "relax" the alias analysis phase. This is controlled by using the -Xrelaxalias=<n> switch, where `n` is:

- `0` Assume that code does not follow the C99 strict aliasing rules. Be more conservative.
- `1` Relax the alias analysis to assume that type instances do not overlap.
- `2` Relax the alias analysis to assume the C99 aliasing rules.
- `3` Further relax the alias analysis to assume that const and non-const variables do not alias.

A value of `1` should be safe in the majority of circumstances. To allow for more optimal code generation, with -O2 the compiler assumes -Xrelaxalias=2 at this level of optimization, at lower levels of optimization the compiler assumes -Xrelaxalias=1.

Enabling strict aliasing allows the optimizer to assume that pointers to different types do not alias. In the following trivial example, this allows the optimizer to assume that `x` and `y` do not alias due to differing types:

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

Compiled with -O2 -Xrelaxalias=0:

```
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
```

Compiled with -O2 -Xrelaxalias=2:

```
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
```

It is possible to write code that makes this assumption invalid. In order to guarantee optimal code generation, this should be rewritten. The following example breaks the strict aliasing rules:

```
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;
}
```

With -Xrelaxalias=2 the compiler is allowed to assume that z and y do not alias due to differing types so it would be allowed to move the store of the return value to before the stores to `z` leaving the returned value as undefined. It is possible to mix files compiled with different settings of `relaxalias` but in general it is better to change the source code to adhere to the strict alias rules of C99.

In a trivial case like this, the optimizer should be able to recognize that `z` and `y` share the same address and still produce the correct code. However, this is not guaranteed and in more complex cases incorrect code may be generated.

The __restrict keyword

The __restrict keyword can be used to give the optimizer extra information to allow it to be more aggressive. It is a qualifier that can be applied to a pointer to allow the compiler to assume that loads and stores by using that pointer do not alias with any other loads and stores. A typical use of the __restrict keyword might be on incoming arguments of a function. Consider the following:

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

Without any further information, the compiler must assume that `x` and `y` can both possibly point to the same memory location. With `-O2` the compiler generates the following code:

```
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
```

Notice that it must complete the store to x before it can load y in case the store modifies the value of y. By using the __restrict keyword on the incoming parameters, you can guarantee to the optimizer that x and `y` do not alias each other.

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

This generates:

```
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
```

Now it has been able to bring both loads to the top of the function and both stores to the bottom. An even better way to convey the same information to the optimizer without using the __restrict keyword is to unpack the incoming pointers into local variables:

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

This generates exactly the same code as using the __restrict keyword:

```
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
```

Again, this allows the optimizer to bring the loads to the top and stores to the bottom of the function.

Inlining controls

There are three main switches to control inlining in SNC. Making adjustments to these values can yield massive improvements to the size and execution speed of compiled code and is probably the most effective user tuneable optimization strategy available on the PS3. For example, disabling inlining causes the render time in a game sample used by SN Systems to go from 12 μsecs to about 80 μsecs. The main reasons for this dramatic slow down are:

- The use of small C++ member functions found in Object Orientated code found commonly in games.
- The PS3 memory subsystem. A function call causing a cache miss and page miss will incur a penalty of hundreds of cycles potentially dwarfing the cycles used in executing the instructions of the function itself.

On PS3 gains can be achieved inlining quite large functions but this has to be balanced against the increase in code size. The Tuner can be used to identify functions which cause stalls due to cache misses.

Therefore, it is highly recommended that users experiment with these values to find the optimal ones for their code and to consider the profile information from Tuner.

The parameters to these controls express the maximum size of a function in terms of ”instructions”. These are internal compiler instructions and are not necessarily the same as individual processor instructions. With `-O2` and higher, functions which fit certain size criteria are automatically inlined.

The key criterion is function size and this is controlled as follows:

-Xautoinlinesize

This switch limits the maximum size of functions that are automatically inlined by the compiler without being marked as inline in the source code. This does not apply to implicitly inline functions such as C++ methods defined inside classes in header files (see -Xinlinesize).

The switch is used as -Xautoinlinesize=<n> where n is:

- `0` No automatic inlining.
- `64` The default value.
- `<n>` Allow automatic inlining of unmarked functions up to a maximum size of `<n>` instructions.

-Xinlinesize

This switch limits the maximum size of implicitly inline functions that are inlined by the compiler. Implicitly inline functions include C++ methods defined inside classes in header files.

The switch is used as -Xinlinesize=<n> where n is:

- `0` No implicit inlining.
- `384` The default value.
- `<n>` Allow automatic inlining of implicitly inline functions up to a maximum size of `<n>` instructions.

-Xinlinemaxsize

This switch controls the maximum amount of inlining into any one function, therefore limiting the size of single functions with automatic inlining enabled.

The switch is used as `-Xinlinemaxsize=<n>` where `n` is:

- `0` No inlining.
- `5000` The default value.
- `<n>` Allow inlining into functions up to a maximum size of `<n>` instructions.

Forced inlining

SNC supports forced inlining through the use of __attribute__((always_inline)). Leaf functions such as low-level math functions (especially where -Xfastmath is used) should generally be marked as always_inline. This ensures that SNC inlines the functions *regardless* of any other inlining settings; such functions are inlined even with -O0.

For example:

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

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

6: PPU-specific tuning options

In the previous section, more generic compiler options for optimization were considered. This section examines compiler options more relevant to programming for the PPU.

VMX register use

On the PPU, there are four main register classes:

- Integer registers
- Floating point registers
- VMX registers
- Condition registers

Converting between these register classes is very expensive, usually in the order of 30-70 cycles. This often means storing to memory, waiting for the data to traverse the store pipeline which is very deep, and then loading back in through the deep load pipeline – another instance of the load/hit/store problem.

To circumvent these costs, SNC has the ability to perform many calculations just by using the VMX processor. The VMX can perform almost any calculation that the PPU can perform except for memory addressing and double precision floating point, which is rarely used in games.

The PS3 PPU ABI uses the FPU to pass floating point parameters, rather than the VMX. However, in the right conditions, SNC can use the VMX registers (effectively locally breaking the ABI) such as when the source code for a called function is available to the caller or the function in question has static scope. This is an additional benefit of "unity" builds where many modules are compiled together as a single (large) file. In this case the compiler will be more likely to have the information needed to perform this optimization for example by having the body of called functions available.

To enable the VMX conversion, it is beneficial to mark all your math functions with __attribute__((always_inline)) to avoid call boundaries that break the conversion.

Floating point control (-Xfastmath)

The –Xfastmath switch allows a number of floating point optimizations which have a significant impact on code performance. This is not enabled by default as the resulting code does not strictly conform to the IEEE standard. However, in general this is not a significant problem.

-Xfastmath=0 Conservative setting. Safe, low-performance floating point.

-Xfastmath=1 Allow conversion of floating point expressions to VMX and use of vector load/store instructions such as lvlx.

Enable branchless fsel selections.

The fastmath option can help avoid the load/hit/store problem on the PPU PS3 architecture by using the VMX unit to avoid memory access. This occurs, for example, when float to int and int to float conversions are carried out. Without fastmath, these conversions cause memory accesses and often load/hit/store penalties.

However, be careful of using –Xfastmath with data residing in video memory. The base address of video memory is available via the macro `RSX_FB_BASE_ADDR` and in the current SDK the base address is `0xC0000000` but this may change in future revisions of the SDK. This memory region has more strict alignment constraints than standard memory and alignment traps may occur with the vector load and store instructions (lvlx etc.) which are used for float/int conversions with fastmath. This can be avoided if the variables resident in video memory are marked as volatile in which case the vector instructions are not used but with a loss in performance. In passing, accessing video memory in this manner should generally be avoided because it implies a large performance penalty.

```
//
// 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
```

Integer to floating point conversions

Integer to floating point conversions can be performed by using VMX registers. This is several orders of magnitude faster than just by using the floating point unit. This is also the case with floating point to integer conversions.

```
//
// 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
```

Sign extend removal (-Xassumecorrectsign)

The -Xassumecorrectsign option controls the removal of sign extends from 32 bits to 64 bits, i.e. making the upper 32 bits copies of the sign bit. It has two settings: safe and aggressive. The aggressive setting can be used if the upper bits of a 32-bit signed or unsigned value (held in a register) are simply copies of the sign bit. (recall the registers are 64-bits wide). This assumption can be incorrect if computations overflow or if pointers are cast to 32 bit ints thus using the upper 32 bits to contain meaningful data and are not simply copies of the sign bit.

Use of the aggressive setting can have a significant impact on code size and performance.

The option is used as follows:

-Xassumecorrectsign=0   Remove only “safe” sign extends where the compiler can guarantee that the upper 32 bits are copies of the sign bit.

-Xassumecorrectsign=1   Assume no arithmetic overflows and no unchecked conversions from 64 bit types specifically pointer types.

```
//
// 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
```

Floating point comparisons (-Xuseintcomp)[5]

The -Xuseintcmp option allows the rewriting of floating point compares, absolutes, and negates as integer expressions. The floating point unit on the PPU has a very long latency (> 20 cycles) and floating point compares usually result in a pipeline flush that may last more than 50 cycles. The switch has two settings: safe and aggressive. The aggressive mode does not handle NANs, but does handle negative zero.

-Xuseintcmp=0	Safe, do not convert compares.
-Xuseintcmp=1	Aggressive, convert compares to integer ops.

```
//
// 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
```

[5] Feature only available in version 270.1 and later

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

Make functions static

SNC is able to make certain ABI optimizations on functions that are static and do not have their address taken for external use. Any functions that are not accessed from outside their compilation unit should be marked as static to allow these optimizations to take place.

For example:

```
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_;
}
```

... generates the following at `-O2`:

```
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
```

However, by simply marking the function getVec() as static, the compiler is able to apply an ABI optimization and return the wrapper struct Vector4 in registers as if it had been tagged with the vecreturn attribute:

```
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():
```

Branchless optimizations[6]

The -Xbranchless switch uses the “branchless” form of comparison on the target machine wherever possible. This avoids flushing the pipeline when the branch is mispredicted and also helps simplify the structure of loops for further optimization opportunities.

On the PPU, there is no conditional move on the integer unit, so this switch is generally off by default to avoid bloating cold code.

There are three settings:

-Xbranchless=0	Off.
-Xbranchless=1	On for ternary operators only, for example a > b ? a : b.
-Xbranchless=2	On for all possible integer comparisons.

```
//
// 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
```

[6] Feature only available in version 270.1 and later

```
        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
```

Loop unrolling (unrollssa) [7]

Many loops can be converted to single basic block loops by using the -Xunrollssa switch. This option takes a parameter that indicates the final size of the loop(s) in the instructions. However, because further optimizations usually happen, the final number of instructions are much smaller so it is worth specifying a large number.

-Xunrollssa=0	Do not convert loops.
-Xunrollssa=10	Only convert very small loops.
-Xunrollssa=30	Convert larger loops.
-Xunrollssa=100	Extreme loop unrolling, unlikely to be useful for real code but may be of use for benchmarks and other small examples.

Remember that if you make a large setting, then your cold code runs slower because each cold instruction fetched costs about twenty cycles.

```
//
// 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
          # branch after branch delay
```

```
# 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
        bne    0, ..L.2 # bc 4,2
```

[7] Feature only available in version 270.1 and later

```
        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
```

```
          # 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: Coding styles for the PPU

This section considers some general points about programming for the PPU and includes of a set of code examples demonstrating key points about the PPU.

Transfers between register classes

Conversions between the main data register classes (scalar, floating point and vector) are costly because the conversions involve memory access. The following code is a simple example:

```
//
// 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 );
}
```

However, in certain circumstances, `float/int` conversions can be handled efficiently if the VMX registers are used as follows:

```
//
// 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;
}
```

Using inlined constants

This is another idiom that can be used to avoid costly memory access:

```
//
// 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 instruction
   c = (vector unsigned int)( 1 );
   d = (vector unsigned int){ 0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f };
}
```

Use of unions for vector casts

Many code samples use unions for casts between types. When the casts are to and from vector types, it is more efficient to use explicit casts rather than the union idiom. The compiler can identify the pattern and use of the vector unit. Note that the fastmath option must be enabled.

```
//
// 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 };
}
```

Latency of vector compare (vec_all_* and vec_any_*)

The vector unit has a deep pipeline so there is a long latency before the results of a vector operation, such as the Altivec intrinsic functions, are available. This can be a problem, for example, in a loop where the test affects the exit conditions. If possible, the code needs to be re-worked to avoid the use of the condition.

```
//
// 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);
}
```

Use pass by value rather than pass by reference

Parameters which are passed by reference (address) tend to cause load/hit/store penalties. Passing large parameters by value is more inefficient, but for scalar and vector types it is always beneficial.

```
//
// 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: Use of SN linker

Fake-signed ELF

The SN linker has the ability to natively produce output to the fake-signed ELF (FSELF) format. The FSELF format is suitable for direct use on the PS3 Reference Tool and Debugging Station, avoiding the need to run the make_fself program (provided in the PS3 SDK) on the linker output. In addition, the internal FSELF output of SN linker is noticeably quicker than the make_fself program, reducing project build time.

FSELF output is enabled by passing the --oformat=fself command-line switch. Note that when this is invoked by using the SN compiler driver, this option becomes -Wl,--oformat=fself.

The SN linker has a number of important generic and PS3 specific optimizations which have a large impact on game performance, but more importantly game size. As with the compiler options, this section covers the generic options and then the (PS3) architecture specific optimizations.

Dead stripping with the SN linker

Introduction

Given the size of modern projects, it is difficult for programmers to manually eliminate unused code and data, and the C/C++ compilation model means that the compiler often emits definitions that are not used in the final executable. The SN linker provides the capability to remove unused (or ”dead”) code and data from a program during the link phase, reducing the overall image size.

The basic approach is to determine the static reference graph of the program and analyze it to determine unused or duplicated elements. This analysis affects the performance of the linker to varying degrees. As such, the choice to enable dead stripping may depend on project size, perhaps being reserved for non-debug builds. However, many developers choose to enable dead stripping for all builds, finding that the memory regained makes the trade-off worthwhile.

All SN linker command-line options detailed below should be preceded by `-Wl`, as with the GCC toolchain, when invoking the linker by using the SN compiler driver. If using VSI (via Visual Studio) the commands should be added as “Additional Options” for the linker in the ProDG VSI Project properties.

Dead-code stripping

In dead-code stripping mode, the executable code is analysed to determine unused functions. In addition, analysis of the table of contents (TOC) section is enabled, allowing the TOC data introduced by statically linked non-SNC built libraries to be reduced. This option is the fastest of the dead-stripping options. As a rough estimate, this process adds roughly fifty percent to the link time.

Dead-code stripping is enabled by using the --strip-unused either on the command line or as an "Additional Options" for the Linker in the Project Properties if using VSI.

Dead-code and dead-data stripping

In the dead-code and dead-data stripping mode, the linker analyses data sections in addition to the executable code to allow removal of both unused code and global data. Since code and data interrelationships are also analysed, dead-data stripping can occur if dead code references otherwise unused data and vice-versa. This mode adds relatively little overhead beyond dead-code stripping alone and is therefore an excellent compromise between build time increase and size reduction.

Dead-code and dead-data stripping is enabled by using the --strip-unused-data command-line switch or via VSI in the "Additional Options" menu for the Linker.

Code and data de-duplication

This mode discovers and removes duplicated objects, such as functions consisting of the same sequences of instructions or read-only data with the same values. Whilst de-duplication can reduce the image size further than dead stripping alone, it may harm cache hit rates since it tends to reduce locality. Furthermore, de-duplication is the slowest of the dead-stripping options.

De-duplication is enabled by passing the --strip-duplicates command-line switch in combination with either --strip-unused or --strip-unused-data. The choice of --strip-unused or --strip-unused-data governs the dead stripping mode in addition to the scope of de-duplication. If using VSI, these options are specified in the ProDG VSI Project Properties menu.

Reporting

The linker can produce a report detailing the savings from the use of dead-stripping options. When linking without dead-stripping is enabled, the report shows the expected size savings. Otherwise, the report shows which objects were stripped or de-duplicated, along with the corresponding size savings. The report also shows which objects have not been stripped and indicates why, which can prove useful when trying to determine why something was not stripped or simply when checking how a function or object is used. Further details are available in the SN linker manual.

The dead-stripping report is enabled by using the --strip-report=filename command-line switch.

Dead stripping and debug data

In the majority of cases, dead stripping preserves the correspondence between the executable and the debug data. However, occasionally minor problems may be apparent. Such problems are due to the transformations performed on the code and difficulty of expressing them in the debug data. De-duplication is prone to this since multiple source functions map to the same object code, potentially causing problems for programmers and debugging tools.

9: PPU-specific optimizations

”No TOC restore” mode

Background

The PS3 PPU ABI prescribes the use of a TOC for accessing global data in C/C++ programs. The TOC itself is a list of 32-bit addresses corresponding to the global data items. However, the method of accessing the TOC limits its size to 64 KB which can necessitate multiple TOCs (also known as *TOC regions*) within a single program. Since the TOC pointer is made available by using a general-purpose register, the PS3 PPU ABI requires that this *TOC register* is set correctly before a function call and restored afterwards. The ABI mandates that function calls are followed by a *nop* instruction, which allows the linker to patch in code to restore the TOC register. Furthermore, calls by using function pointers or virtual functions require boilerplate preamble in order to set the TOC register correctly.

SNC’s approach

SNC does not use the TOC to access global data. Instead, it emits a sequence of two instructions which generate the required data address, in collaboration with the linker. Despite a slight increase in code size, this choice benefits the code by reducing the size of the TOC section in addition to avoiding the extra indirection required to access data by using the TOC, improving cache usage and reducing average latency on global data access.

However, the ABI still requires the trailing *nop* instruction for interoperability with other code, for example, GCC-built code residing within PRX libraries. However, this requirement is usually excessive for game code since much of the code is built together from the source. This often leads to large numbers of unnecessary *nop* instructions, and examination of real object code shows that a typical game contains between **one thousand** and **one hundred thousand** *nop* instructions. This indicates that up to several hundred kilobytes of the code size may be contributed by unnecessary *TOC-restore* placeholder instructions. The compiler may still generate other *nop* instructions unrelated to this use for the ABI in order to align code for architectural or performance reasons.

”No TOC restore” mode

”No TOC restore” mode is available when using SNC and the SN linker together. It allows the omission of TOC-restore placeholder instructions and reduces the instruction sequence required to call by using a function pointer or virtual function, leading to more compact code. For most code, ”no TOC restore” mode provides improvements to the code size with no negative impact.

”No TOC restore” mode is enabled by invoking the compiler driver as follows:

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

Alternatively, if compiling and linking separately, issue the following commands:

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

In this mode, the compiler assumes that the targets for all function calls exist within the same TOC region and it suppresses any code that facilitates saving and restoring the TOC register. At the same time, the linker enforces the presence of a single TOC region in the final executable. If the linker detects that multiple TOC regions are required due to TOC contributions from statically linked libraries, it raises an error and fails the link phase. At this point, reducing the amount of TOC data or disabling ”no TOC restore” mode are the only two courses of action.

The assumption of a single TOC region is completely safe even in the presence of calls to PRX libraries since they occur by using linker-inserted stub functions. In ”no TOC restore” mode, the linker rewrites the stub functions to modify and restore the TOC pointer as necessary. Note that code making very large numbers of calls to PRX libraries may show smaller code size improvement in ”no TOC restore” mode since the replacement PRX stub function is slightly larger than the original. For more information, please see the SN linker manual.

Usage restrictions

In principle, the PS3 PPU ABI guarantees the safety of ”no TOC restore” mode. However, as with many optimizations, real code shows corner cases that ”no TOC restore” mode does not handle correctly. These cases are fortunately both rare and simple to avoid.

These corner cases arise because C/C++ allows calls into a PRX library, and so into a different TOC region, without realising. The two cases in which this may happen are function pointer and virtual function calls. If such a call occurs in code compiled in ”no TOC restore” mode, runtime failure occurs at the first use of the TOC. However, it is safe to call from PRX code into ”no TOC restore” code in this manner.

Unfortunately, this problem cannot be diagnosed at build time due to uncertainty in the call target when dealing with indirect function calls. For the interested reader, the SN linker manual details these cases more thoroughly. As a further point of interest, PRX libraries built with the SNC toolchain in ”no TOC restore” mode eliminate this restriction. However, all system PRX libraries are currently built with the GCC toolchain.

In summary, avoid taking pointers to functions in PRX libraries or using C++ classes containing virtual functions defined in PRX libraries when using ”no TOC restore” mode.

Workarounds for PRX library code

If calls into PRX libraries are unavoidable in any of the previously mentioned cases, it is still possible to use ”no TOC restore” mode to gain improvements in the majority of the code base.

SNC provides #pragma directives for controlling ”no TOC restore” mode on a per-function basis. With these directives, the marked functions behave as if ”no TOC restore” mode is disabled. The following example demonstrates this feature:

```
#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
```

The use of #pragma noinline in the previous example prevents the compiler from inlining from function, which negates the disablement of ”no TOC restore” mode.[8]

[8] Note if using an earlier compiler than 270.1 The Cell GCM library contains some code that is unsafe in ”no TOC restore” mode. The macro CELL_GCM_SNC_NOTOCRESTORE_2 should be defined during compilation when using the library. This macro definition replaces the hazardous code with a safe alternative. In SNC 270.1 this is not necessary.