Under the Compiler's Hood: Supercharge Your PLAYSTATION®3 (PS3™) Code

Andy Thomason: SN Systems Ltd

andyt@snsys.com

Introduction

This paper is presented in two parts.

The first part is a brief description some of the internal workings of an optimizing compiler. Much of this will be familiar material to anyone with a computer science degree.

The second part uses the language established in the first part to illustrate some simple methods that can be used to rewrite poorly performing code.

The PS3™ with its cell processor incorporating a dual thread Power PC core and up to eight number crunching SPUs presents a particular challenge.

The chip's designers have made some decisions that enable the cell to run at a very high clock rate and have excellent peak performance, but in doing this they have sacrificed some of the "comfort" factor present in workstation processors. In this paper we suggest some ways to avoid the pitfalls.

There is no hard and fast rule for improving the performance of compiled code. Different compilers will work better with different code and it is up to the programmer to find methods that work on a wide range of compilers.

Benchmarks are a useful measure of a compilers performance but are restricted to "hot" code that is frequently executed and do not reflect the demands on caches that real projects make.

Compiler writers work hard to improve the code generation in specific areas based on experience gained from compiling hundreds of projects. If a programmer can isolate a very specific instance of poor performance and suggest a better alternative, this can often be quickly integrated into the compiler.

Optimizations often lie in two groups. One group of optimizations enable the programmer to write elegant code without hand optimizing every function, the other group simply corrects errors that inexperienced programmers introduce. Clearly in games programming we wish to focus on the former, but sadly we cannot neglect the latter.

By learning to communicate our intentions to the compiler concisely, it is possible for a programmer to learn a set of rules that will guarantee good code performance with a wide range of compilers.

We hope that some of these rules will be useful to you.

Part 1: Compiler internals

Trees & parsing

The first stage in compiling code is to read the text of the source code and convert this to a tree structure in the compiler.

In the past this was usually done with a parser generated from a grammar description such as YACC or BISON, but most recent compilers use a hand-coded recursive-descent parser that is easier to bug fix, as well as potentially faster.

In GCC, the front end produces trees in a format called GENERIC, while the EDG front end used in SNC produces a tree of structures.

It is here that most of the language-specific features are dealt with and many optimizations are performed while the code is in tree form.

Basic blocks

At some point in the compilation process trees are converted to a pseudo assembler broken up into “Basic Blocks”.

Each basic block represents a section of the final assembler code between labels and branches.

For example, in example 1 there are three basic blocks labelled with assignments to the variable bb. This function will translate into the pseudo code shown below.

```
int f( int x )        BB1:
{                       copy bb, 1
  int bb = 1;           bne x, 0, BB3
  if( x == 0 )        BB2:
  {                     copy bb, 2
    bb = 2;           BB3:
  }                     copy bb, 3
  bb = 3;               return
  return bb;
}
```

Example 1: a function with three basic blocks

In GCC the pseudo code is called GIMPLE. This pseudo code or “*intermediate representation*” allows compilers to make optimizations for many targets.

Once the code is in basic-block form. The blocks themselves form a “*control flow graph*” that can be optimized to combine, split and re-route control flow.

As optimizations work best on big basic blocks, they are often expanded by “*tail duplication*” shown in Example 1a to avoid re-merging.

Another popular method is to “*unroll*” loops so that more space is available for scheduling

```
BB1:
  copy bb, 1
  bne x, 0, BB3
BB2:
  copy bb, 2
  copy bb, 3
  return

BB3:
  copy bb, 3
  return
```

Example 1a: tail duplication

Data flow analysis

Modern compilers use "*single static assignment*" form to represent data flow within a function.

Example 2 shows how each assignment to the variable "x" is treated as the creation of a new variable.

The pseudo-op "phi" is used to combine the results from two merging basic blocks. This instruction is only there to make sure that both basic blocks give their result in the same register and as a signal to the optimizer that this value has more than one definition.

This means that each expression in the function can be traced back through a tree of inputs.

With loops, the phi node may loop back on itself. This is used to detect loop iteration variables.

```
int f( int x )
{
  if( x == 0 )
  {
    x = 2;
  } else
  {
    x = 3;
  }
  return x;
}

BB1:
  bne x, 0, BB3
BB2:
  copy x1, 2
  b BB4
BB3:
  copy x2, 3
BB4:
  phi x3, x1, x2
  return x3

x3 = phi( 2, 3 )
```

Example 2: single static assignment form

Alias analysis

Alias analysis is used to determine if loads from the same address can be combined, if load/store pairs are redundant or if memory operations can be re-ordered to reduce latency.

At its heart alias analysis just answers the question "can these two memory addresses overlap".

Example 3 shows a function of two pointers and various alias analysis decisions that have been made.

Optimizations may be made if it is decided that two pointers do not interfere. Otherwise the loads and stores can not be removed.

Awareness of alias analysis is the simplest and best way to improve your code performance.

```
void f( char *p, int *d )
{
  d[ 0 ] = 1;
  int a = *p; // *p (char ) does not alias d[ n ] (int)
  d[ 1 ] = 2;
  int x = 2;  // d[ n ] does not alias x (formal vs stack)
  d[ 2 ] = a;
  g( &x );    // call, x may have been modified
  d[ 3 ] = x;
}
```

Example 3: Alias analysis

Invariant code motion

As an example of the optimizations performed by a compiler, invariant code motion moves unchanging expressions out of a loop.

Again, alias analysis performs a crucial role in determining if loads and stores can be migrated.

Optimization of inner loops will often give the greatest benefits in game code.

```
void f( int a, int b, int *p, short *q )
{
  for( unsigned i = 0; i != 100; ++i )
  {
    // load from q doesn't alias p so we can move it to before the loop
    p[ i*2 ] = q[ 0 ] + a;

    // a + b is invariant, we can move it out of the loop
    p[ i*2 + 1 ] = a + b;

    // store to q[ 1 ] is invariant, we can move it to after the loop
    q[ 1 ] = a;
  }
}
```

Example 4: Invariant code motion.

Load/Store elimination

If we can determine that a store-load pair is to the same address and that no intervening store effects the result in memory, the load may be removed.

On the PS3™ this is a crucial optimization as the PPU processor incurs a “load hit store” penalty if the load is within twenty cycles of the store. These

penalties are common with modern processors which are optimized to work with values in registers.

Equally, if a value is stored back to the same address, the store may be removed.

If it is determined that no loads remain from a variable, then all the stores to that variable may be removed.

Copy and constant propagation

These optimizations combine expressions across basic block boundaries. Example 5 shows a typical example.

The constant propagation algorithm uses a set of rules to combine arithmetic operations to reduce the number of instructions required to execute them.

Useful tips on the PS3™ and other processors to help constant propagation are:

- Use unsigned values where possible, avoiding sign extend instructions
- Use enums for integer constants (static const may be evaluated late)
- Avoid divides and pointer difference
- Evaluate expressions in the same basic block

```
int f( int a )
{
  a++;
  a++;
  a++;
  return a;
}

->

int f( int a )
{
  return a + 3;
}
```

Example 5: copy and constant propagation.

Scheduling

The PS3™ PPU, like most modern processors, has several cycles of latency before the results of executing an instruction may be re-used.

Floating point calculations are the worst, with tens of cycles of latency possible.

By having long pipelines, it is possible to make a processor that has fewer gates at each stage with latches in between. The processor can complete many stages in a single cycle, like workers on a production line, and with fewer gates for each stage, the results are ready sooner. Hence the clock rate may be increased and the theoretical throughput of the processor increased.

If the basic blocks are big enough and we are not plagued by aliasing problems, the compiler can re-order the instructions in a basic block so that the time spent waiting for instructions to complete is used for other calculations.

The time needed to execute the longest expression tree is called the “critical path”. If instruction latency is dominant, the length of the critical path determines the speed of the code.

Example 6 shows a simple scheduling example. Here the “mul” instruction takes several cycles to complete. Therefore we move the second store to before the first so that it can use some of the dead space after the multiply. Now we only take four cycles to execute, rather than five.

In practice scheduling also takes account of instruction pairing to make the best use of multi-instruction issue on the PPU and SPU.

```
p[ 0 ] = a * b;
p[ 1 ] = c ;

mul t1, a, b            mul t1, a, b
# stall                 store c, p, 1
# stall                 # stall
store t1, p, 0          store t1, p, 0
store c, p, 1
```

Example 6: scheduling

Register allocation

Register allocation assigns real machine registers to the expression results used by the pseudo code.

In the process, it may be necessary to save registers in memory, a process called “spilling”.

Global register allocation allocates registers that are used across basic block boundaries.

Local register allocation allocates registers within basic blocks.

Register allocation is complicated by “partial write” instructions which use one of their inputs as an output register. On the PPU we have the “update” instructions such as stdu and lbzu which write back to their address register and rlwimi which inserts data into its source register.

Live ranges represent the “lifetime” of an expression. Whilst most expressions will have a single definition and use, some expressions may be used many times throughout the function.

A live range can be “live in” or “live out” if it crosses the boundaries (edges) of a basic block.

We can use scoping to limit the lifetimes of variables, especially in c++, where destructors may keep values alive for longer than expected. This will free up registers that may be spilt as well as making code more readable. As with all optimizations, inspect the code that is generated to see if this improves spills.

```
void f()                          void f()
{                                 {
  MyClass x;                        {
  y = x;                              MyClass x;
  ... many lines of code              y = x;
  // here we have an                  // destructor call
  // implicit destructor            }
  // call on x                      ... many lines of code
}                                 }
```

Example 7: limiting lifetimes through scoping

Profile driven optimizations

A fast developing field of compiler optimization lies in the use of profile information recovered from running the code with special settings.

We can divide basic blocks or whole functions into “hot” and “cold” sections.

Hot code is code that is executed frequently, like a matrix stack calculation loop or a software shader.

Cold code is executed infrequently, but has an impact on instruction cache usage.

For hot code it is worth using more instructions to get faster execution. So we can use more aggressive inlining, loop unrolling, cache hints, branch hints and tail duplication of “if-else” statements.

For cold code we want to use fewer instructions. This code will pollute the instruction cache and slow down other code, so it is important to make it as small as possible and to avoid sharing cache lines with hot code.

On GCC the “gcov” tool is used to count the use of basic blocks and SNC will be supporting a PS3™ -specific tool.

Part 2: How to write efficient C/C++ code

Maximising basic block sizes

If your code is “hot” then you probably want to work to increase the size of basic blocks by using fewer “if” statements, && and || operators.

The PPU has a typical cost of eight cycles for a branch instruction, so bigger basic blocks mean fewer branches.

Calls interrupt basic blocks and cost dearly for branches and register saves. Using inlining avoids this overhead.

Using __attribute__((always_inline)) on small functions is a big winner, especially on math libraries.

Try to avoid separated “if” statements. If you have to use “if” statements, the space between them represents a small basic block that could be combined with one above. The compiler may do this automatically, but aliasing may prevent the optimization from happening.

The PS3™ PPU, like most modern processors, has a very high latency on floating point compares. It is therefore a good idea to minimize the use of floating point expressions in control statements. It is usually best to calculate all expressions and then select the one you want using the select operator which can be optimized.

On SNC, many simple branches may be removed using the –Xbranchless option.

```
// bad
if( p[ 0 ] == 1 && p[ 1 ] == 2 ){}

// good
int a = p[ 0 ], b = p[ 1 ];
if( a == 1 && b == 2 ) {}

tiny bb containing load of p[ 1 ] created
```

```
// bad
// do something
if( x == 0 ) { }
// do something
if( y == 0 ) { }

// good
// do something
if( x == 0 ) { }
if( y == 0 ) { }

tiny bb containing second do something created
```

```
// bad
if( x == 0 ) { a = 1; } else { a = 2; }

// good
a = 2;
if( x == 0 ) { a = 1; }

tiny bb containing a = 2 created, extra branch required
```

```
// bad
if( x > 0.1f ) { a = 1.0f; } else { a = 2.0f; }

// good
a = x > 0.1f ? 1.0f : 2.0f;

all-floating point select expressions may be reduced to fsel
```

Example 8: good and bad "if" statements

Minimising effects of latency

If you use similar expressions in the same basic block, high latency VMX and FPU instructions will be able to be interleaved. An example of this is calculating a trig function using an inline function. Calling the inline function once will result in large gaps between instructions as the latencies are high. Call the same function four times and the gaps will be filled giving you four times the throughput.

Be careful not to increase register pressure too much, however. Really big basic blocks can spill registers if lifetimes are too long.

The SN pipeline analyser incorporated into the debugger, SNBIN and the tuner can be used to show stalls in compiled code. By tweaking the code, it is possible to minimise these latencies.

Using two threads on the PPU is a big latency winner, more about this later.

On the PPU, avoid re-loading values that have just been stored into memory. The store unit will have to be flushed resulting in a load-hit-store penalty of fifty cycles or more. Inlining and helping alias analysis to function by copying values into local variables is a big winner here.

Try to avoid using memory loads in multi-part "if" conditions. This will enable the conversion of the multiple conditional branches into boolean expressions. On the PPU, there is a set of instructions like "crand", "cror" to combine condition codes that can only be used if expressions have no side effects.

```
// bad
if( p[ 0 ] == 1 && p[ 1 ] == 2 ){}

// good
int a = p[ 0 ], b = p[ 1 ];
if( a == 1 && b == 2 ) {}
```

Example 9: good and bad "if" statement conditions

Avoiding aliasing

As we have mentioned before, if there are too many loads and stores in your resulting code, then there is probably a problem with alias analysis.

You can use the __restrict keyword to tell the compiler that a pointer will not alias, but too many restricts look ugly and may not be portable.

Unpacking pointer references into local variables is a more portable and readable alternative that allows loads and stores to be migrated for optimal efficiency. We can use variable unpacking to add more semantic meaning to the code as the variable names can be descriptive.

Type conversions and unions

The PS3™ PPU has three distinct arithmetic register sets; int, float and vector. Moving results between register sets is very expensive.

If you are intending to move between the sets, use a union on the stack frame. With optimization on, it may be possible for the compiler to backtrack the calculation so that the whole thing is performed in one register set.

Example 10 shows the correct use of a union to perform a type conversion. The variable "u" is an auto variable that will be represented in memory. In practice, this union will simply represent two different classes of register, in this case float and vector.

It may be possible or the compiler to carry out the entire operation in VMX registers; if the destination is a dereferenced pointer, for example.

```
static inline float f( vector float x )
{
  union {  float f[ 4 ]; vector float v; } u;
  u.v = x;
  return u.f[ 1 ];
}
```

Example 10: use of a union on the stack frame to signal a register type change.

Although it is convenient to mix register classes in unions that are accessed by pointers, alias analysis my not be able to optimize to the same extent, resulting in a memory store-load pair being generated.

Another problem with using unions in pointer-referenced classes is the ambiguity over how to copy the structure as a whole.

Passing by value becomes much harder with structures containing unions with inevitable uses of store-load pairs for register class conversion.

Example 11 shows the incorrect use of a union in a class. Many poorly-performing math libraries are written this way.

```
struct Naïve
{
  union {  float f[ 4 ]; vector float v; } u;
};

float f( Naive &x )
{
  return x.u.f[ 0 ];
}
```

Example 11: incorrect use of a union in a structure or class.

As the vector type is the most flexible, being able to perform all the operations of the other two classes with the exception of memory indexing, it is best to use only this type in a class.

```
struct Smart
{
  vector float v;
  float x()
  {
    union {  float f[ 4 ]; vector float v; } u;
    u.v = v;
    return u.f[ 0 ];
  }
};

float f( Smart &x )
{
  return x.x();
}
```

Example 11: better use of a union, struct copies will use vector float

PS3™ intrinsics vs. inline assembler

On the whole, it is much better to use intrinsics on the PS3™ than to use inline assembler.

For a start, intrinsics are schedulable meaning that they integrate efficiently with the code around them.

Alias analysis will be very conservative in the face of inline assembler, causing many more loads and stores to be left in the code.

Intrinsics are more portable. On the PS3™, many intrinsics work the same way on both the SPU and the PPU compilers.

The only place where inline asm is a win is where instructions are not supported by intrinsics and for accessing arcane features of the assembler like the “.incbin” directive.

Vector classes dos and don'ts

It is a sad fact that most Vector Math classes used in console games have origins on PCs. The result is that they usually break the rules in the section *Type conversions and unions*.

The PPU compilers support efficient pass-by-value for classes containing exactly one data member of VMX type.

It may be worth having "storage" classes and "working" classes with the storage versions being simple binary containers for packing the data into memory used in structures and the working versions performing the math functions on auto variables in registers.

As we have mentioned before, mixing int, float and VMX register classes is a bad idea, so try to do float to int conversions direct to memory and do not use the result for tens of cycles. Example 12 shows such "loop splitting" used to hide the latency of the necessary memory operation.

In the case of example 12, the whole operation could be performed in VMX registers but for the fact that the integer result is used to index a table. The loop must be split to hide the latency.

"Supervectors" are classes containing groups of SIMD registers that can be used to hide latency in complex arithmetic operations.

For example, suppose we wish to calculate the sine and cosine of angles in a particle system. If we use a 16-float supervector composed of four VMX registers, then we can calculate 16 sines simultaneously at maximum clock rate.

As each operation such as add or multiply is performed four times, we have plenty of instructions to fill stall slots. In practice, scheduling will mean that not as many registers need to be used as there are stall slots.

The SN vector math library **SnMathLib** available for the PC, PSP® (PlayStation® Portable) and PS3™ illustrates some of these principles.

```
float input[]; int output[], table[], tmp[];

// bad
for( i = 0; i != 100; ++i )
{
  output[ i ] = table[ (int)( input[ i ] * 2.0f ) ];
}

// good
for( i = 0; i != 100; ++i )
{
  tmp[ i ] = (int)( input[ i ] * 2.0f );
}

for( i = 0; i != 100; ++i )
{
  output[ i ] = table[ tmp[ i ] ];
}
```

Example 12: loop splitting to avoid direct use of float->int conversion

```
float16 angle[ 100 ]; // 1600 values
float16 sine[ 100 ];
float16 cosine[ 100 ];
for( i = 0; i != 100; ++i )
{
  sine[ i ] = sin( angle[ i ] );
  cosine[ i ] = cos( angle[ i ] );
}

# regular vector code (6 stalls, throughput 4)
  vaddfp t1, a1, b1
  # stall
  # stall
  # stall
  vmaddfp s1, a1, t1, zero
  # stall
  # stall
  # stall

# supervector code (no stalls, throughput 16)
  vaddfp t1, a1, b1
  vaddfp t2, a2, b2
  vaddfp t3, a3, b3
  vaddfp t4, a4, b4
  vmaddfp s1, a1, t1, zero
  vmaddfp s2, a2, t2, zero
  vmaddfp s3, a3, t3, zero
  vmaddfp s4, a4, t4, zero
```

Example 13: supervectors. Each function operates on 16 values at a time.

Multithreading effects on PS3™

On the PPU if two threads execute at the same time then instructions will be executed alternately. This effectively halves the latency as the instruction issue for one thread is halved.

This means that the arithmetic instructions will run back-to-back without delay and floating point instructions run in around five cycles.

The SNC compiler supports this behaviour with the –Xthreads=2 switch which divides the latency by two.

If there is a cache miss on one thread, the other can one pick up the slack and get on with processing instructions while one thread waits.

It is a good idea to run your own second thread even if it just consists of “dbcyc” nops. Some single-thread benchmarks show a 30% gain in performance from this simple measure.

Don’t be tempted to run more than two threads on the PPU, however. The thread switch time is not pretty and will break your caches. Do not use the operating sleep function too freely as the same applies.

Finally, use the SPUs as much as possible. This will become easier as advances are made in the compilers. The SPUs are truly awesome number crunching processors.

Virtual function calls and switches

Since their introduction, virtual function calls have proven to be an extremely useful tool for abstracting class behaviour.

However, this comes at a terrible price.

A virtual function call has to fetch the address from memory, often invalidating the dcache, and then make an indirect call which invalidates the icache.

In very cold code this will introduce over a thousand cycles of latency.

So it is best not to use virtual functions to fetch variables from classes; reserve them for very high level functions.

The same applies to switch statements that may use indirect branches also. Be careful to group case values so they are sequential, otherwise the switch will have to be broken into a tree of branches.

Some compilers use the convention that the first case label is executed most frequently which can be used to hint to the compiler. It may be worth maintaining this convention until we fully support it.

Console vs. PC programming

If you are new to console programming then the following advice may be useful, otherwise look away now…

Avoid using malloc in-game. All memory allocation should be done at load time only as malloc is a very slow function. Excessive use of malloc will cause memory fragmentation which is not corrected by virtual memory system on consoles.

Pre-build display lists. The reason that consoles can compete with more powerful workstations is that the GPU (Graphics Processor) is identical and can be fed by pre-built display lists. A PC OpenGL call will take an average of several thousand cycles. If display lists are pre-built, no API calls are necessary.

Lay out your classes and structures so they can be spooled directly from DVD or Blu-Ray. Serialization wastes CPU resources and prevents instant background loads.

Make use of thread synchronization intrinsics. Although it is better to use separate memory spaces for threads, occasionally you will need to communicate between them. Using thread synchronization intrinsics will improve performance for critical sections and message queues.

Do not use global variables. Good general advice for any project. Although they are part of the language they create alias analysis problems and global state is best grouped into one or more “context” classes.

Finally, a recent trend that has huge benefits is to reduce the number of modules in the compilation. Now that parsing is much faster than linking it is sensible to define code in header files and use a single .CPP file. Not only will the code be more efficient, allowing more inlining, but the compilation may be faster! If you don’t believe this you should try it even on big projects.

Special optimizations are available when using this form of compilation using the SNC compiler.

Using SN systems tools to examine your code

If you have been coding for consoles for some time you will be familiar with the SN Systems range of development tools including ProDG, Tuner, ProView and SN-DBS.

The debugger is the obvious first place to start as it will reveal the overall structure of your project. Often simply stopping the debugger at random will show up hotspots in your code.

The SNBIN utility has been enhanced to provide information on ELF files and perform a number of insertion and conversion functions. The pipeline analyzer is available on disassembly views.

Tuner is available for PS3™ and can be used to analyze timing information on a frame-by-frame basis. Functions can be instrumented automatically to give timing details for the call graph on all threads.

Using Tuner we have found many typical hotspots in game code. Unsurprisingly, the largest are usually associated with memory reads with delays at branch targets coming a close second.

New SNC optimizations

SNC has been enhanced with a new *single static assignment* analysis phase.

Single static assignment analysis is used to drive dozens of new optimizations mostly derived from study of typical game project code.

Better constant propagation and memory optimizations reduce instruction counts.

VMX tree replacement is able to replace floating point and integer expression trees with VMX equivalents where necessary.

Auto vectorization does VMX tree replacement on groups of similar operations that can share a single register.

By converting to a “Homogenous form”, floating point compares with long latencies can be executed using integer instructions improving sort functions considerably.

Some additional basic-block level optimizations remove zero and single iteration loops.

SnMathLib

SnMathLib is an example of a vector math library that can be incorporated into game projects for the PC, PSP™ and PS3™. It has been tested on many projects to date and has been modified and adapted by many others.

SnMathLib uses a “scalar” class to avoid float to VMX register conversions.

The library is regularly checked with an extensive test suite for compilation, accuracy and performance.

Errors of trig functions are carefully monitored.

Conclusions

Write big modules with functions containing big basic blocks.

Use inlining to increase basic block size.

Reduce numbers of “if” statements and unpack conditions into local variables.

Avoid conversions between int, float and VMX register classes.

Use the Debugger, Tuner and SNBIN to examine the output of the compiler.

Finally, don’t be a control freak. Try to write simple, understandable code rather than relying on esoteric compiler features to optimize your code. The simpler the code, the easier it is to optimize it. Constantly revise the code until you are proud of it.

Essential reading

Engineering a compiler (Cooper & Torczon)

Wikipedia: http://en.wikipedia.org/wiki/Category:Compilers

GCC internal documentation: http://gcc.gnu.org/onlinedocs/gccint/