SN SYSTEMS
Sony Computer Entertainment Group

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

Understanding your compiler is the key to success in the gaming world.

Supercharge your PS3 game code

- Part 1: Compiler internals.
- Part 2: How to write efficient C/C++ code.

Part 1: Compiler internals

- Trees & parsing
- Basic blocks
- Data flow analysis
- Alias analysis
- Invariant code motion
- Load/Store elimination
- Copy and constant propagation
- Scheduling
- Register allocation
- Profile driven optimisations
- + register allocation & “live ranges not scope are important”

Trees & parsing

```
void f( int *p )
{
  for( int i = 0; i < 6;
++i )
  {
    p[ i ] = 0;
  }
}
```

As a first step the text of the file is converted into a tree structure.

```
TREETOP
   FUNC_HEADER
     FUNCTION
       _Q1fPi
       p
   EXEC_STMT
     BLOCK
       i= I4:0
       WHILE
         i< I4:6
         BLOCK

*(CAST(type_44,p))[i]= I4:0
           i=i+ I4:1
       return NIL
```

Trees & parsing

- Inlining done by merging trees
- Constant folding

```
a + 1 + 2
    -> a + 3
```

- If conversion

```
if( x == 0 ) y = a; else y = b;
    -> y = ( x == 0 ) ? a : b;
```

Basic blocks

```
TREETOP
   FUNC_HEADER
     FUNCTION
       _Q1fPi
       p
   EXEC_STMT
     BLOCK
       i= I4:0
       WHILE
         i< I4:6
         BLOCK
           *(CAST(type_44,p))[i]= I4:0
           i=i+ I4:1
       return NIL
```

Next, the tree is broken into basic blocks.

A basic block is a section of code that contains no branches or labels.

```
BB:1
  i = 0
  *IF (i  < 6) ? goto BB:2  else  goto BB:3

BB:2
  tmp2 = 0
  tmp3 = impy ( i,4 )
  tmp4 = copy4s ( tmp3 )
  store ( tmp2, p, tmp4 )
  tmp5 = iadd ( i,1 )
  tmp5 = copy4s ( tmp5 )
  i = tmp5
  *IF (i  < 6) ? goto BB:2  else goto BB:3

BB:3
  *RETURN
```

Basic blocks

- Assignments translated to loads and stores
- **`if`**, **`while`**, **`switch`** etc. converted to basic block boundaries

example:

```
int f( int *p )
{
  *p = 1;        // store
  int a = *p;    // load
  if( a == 1 )  // condition
  {
    p++;         // expression
  }
  return a * 2; // return expression
}
```

Basic blocks: Unrolling

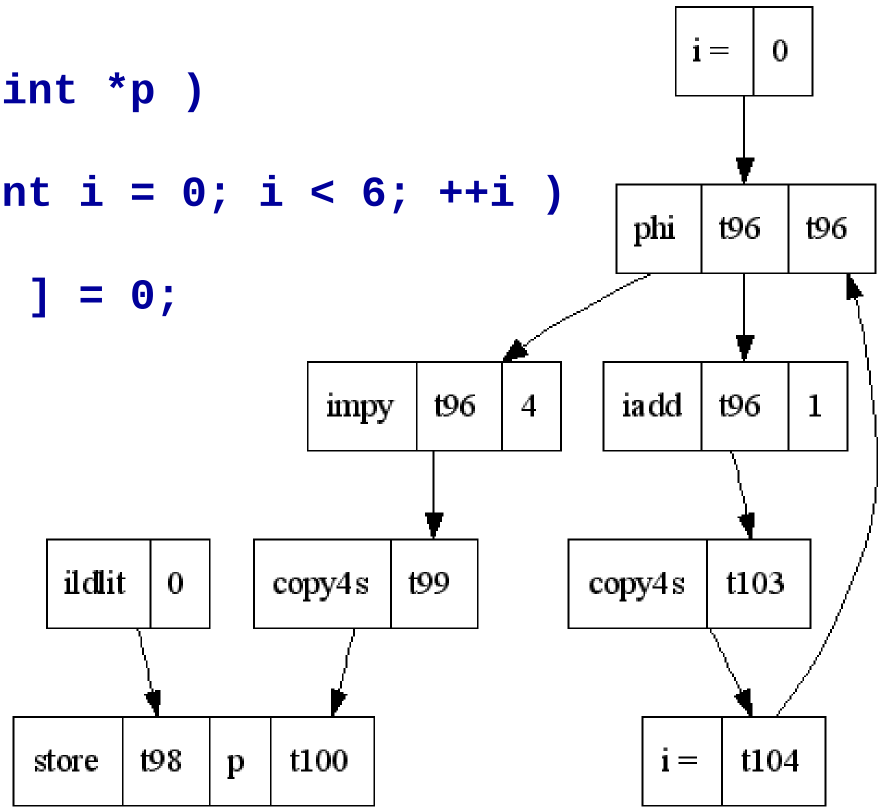
```
// 3 basic blocks

void f( int *p )
{
  for( int i = 0; i != 3; ++i )
  {
    p[ i ] = i;
  }
}
```

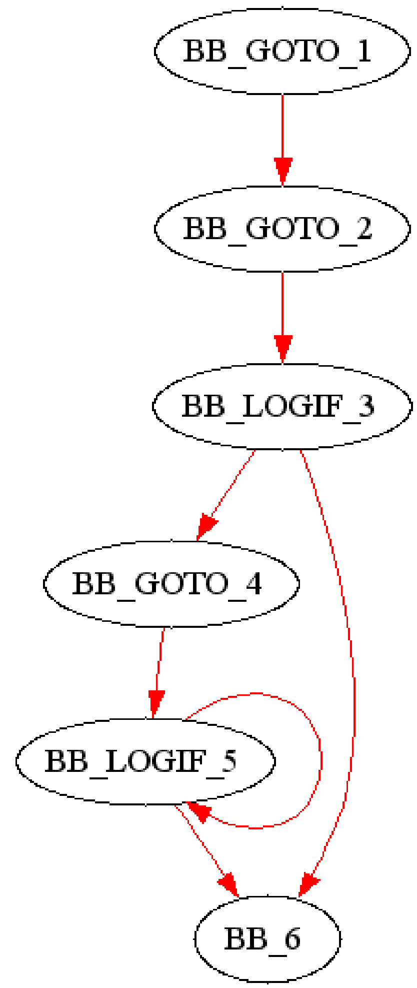
```
// 1 basic block

void f( int *p )
{
  p[ 0 ] = 0;
  p[ 1 ] = 1;
  p[ 2 ] = 2;
}
```

Data flow analysis

Example:

```
void f( int *p )
{
   for( int i = 0; i < 6; ++i )
   {
      p[ i ] = 0;
   }
}
```





Alias analysis

- Tests to see if loads, stores and calls interfere with each other.
- Enables the reordering of loads and stores.
- Enables the elimination of redundant loads and stores.
- Controllable using the __restrict keyword.

Example:

```
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;
}
```

Invariant code motion

- Moves as much code as possible out of loops
- Fewer instructions in loops
- Dependent on aliasing!

Example:

```
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;
  }
}
```

Copy and constant propagation

- Combine assignments and expressions
- Uses fewer instructions

Example:

```
void f( int i, int *p )
{
  // copy propagation, all the same variable
  int a = i;
  int b = a;
  int c = b;

  // constant propagation
  p[ c++ ] = 0; // -> c + 0
  p[ c++ ] = 1; // -> c + 1
  p[ c++ ] = c; // -> c + 2
}
```

Scheduling

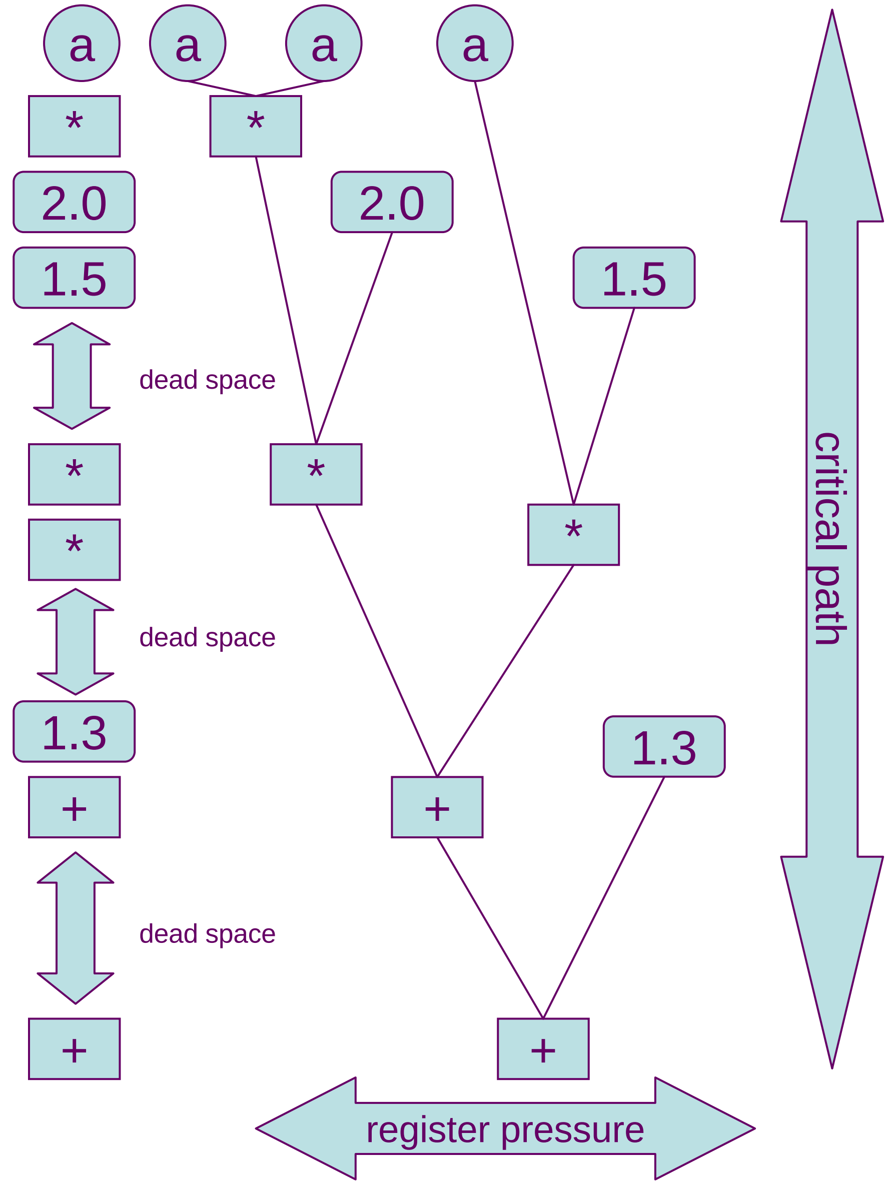
- Re-order instructions to avoid stalls

- FPU/VMX operations take many cycles to complete

- Bad aliasing prevents efficient scheduling

Example:

```
float f( float *p, float a )
{
  return a * a * 2.0f +
    a * 1.5f + 1.3f;
}
```



Register allocation

Expressions allocated a “Local” or “global” register in the function.

Global registers usually in short supply.

Too many registers used lead to “spills” to memory.
Also if address is taken of variable.

Example:

x is in a global register

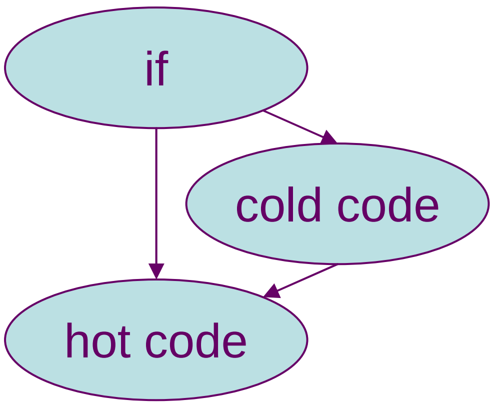
```
x = a;
if( cond ) { /*...*/ }
y = x + 1;
```

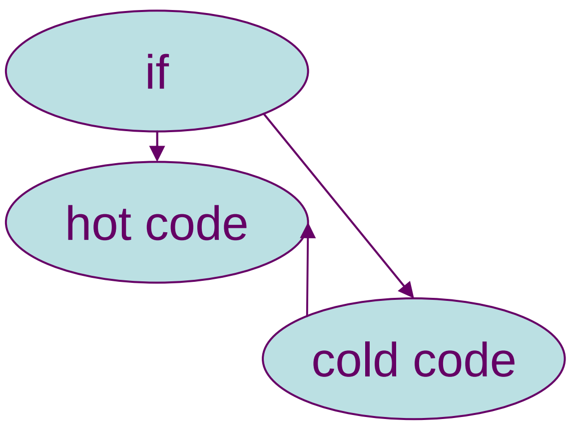
x is in a local register

```
if( cond ) { /*...*/ }
x = a;
y = x + 1;
```

Profile driven optimisations

- Use results of profiling to determine “hot” and “cold” code
- Hot code gets more instructions
  - Inlining
  - Loop unrolling
- Cold code gets fewer instructions
  - Moved away from hot code to prevent icache pollution
- On GCC “gcov” tool





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

- Maximising basic block sizes
- Minimising effects of latency
- Avoiding aliasing
- Type conversions and unions
- PS3 intrinsics vs. inline assembler
- Vector classes dos and don’ts
- Multithreading effects on PS3
- Virtual function calls and switches
- Console vs. PC programming
- Using SN systems tools to examine your code
- new SNC optimizations
- SnMathLib

Maximising basic block sizes

- Inline everything you can and use fewer, larger modules
- Use __attribute__((always_inline)) on small functions
- Be aware of the high latency on floating point compares on PPU
- Even predicted branches are slow on deeply pipelined processors

```
void bad( bool x )
{
  if( x )
  {
    // do something
  }
  // do something
  if( x )
  {
    // do something
  }
}
```

```
void good( bool x )
{
  if( x )
  {
    // do something
    // do something
    // do something
  }
}
```

Minimising effects of latency

- Interleave similar expressions in same basic block

  a0 = b0 * 3.14f + c0 * 1.257f;
  a1 = b1 * 3.14f + c1 * 1.257f;
  a2 = b2 * 3.14f + c2 * 1.257f;

- Use two threads on the PPU

- Load-hit-store on modern processor cores

  a[10] = b;
  c = a[10]; // same address

- Floating point compare

  // try to make this block very big
  if(fabsf(x) < epsilon) {}

- Simplify && and || expressions

```
if( p[0] == 0 && p[1] == 0 )

  lwz
  cmp
  bc     # 2-22 cycles
  lwz
  cmp    # 2-22 cycles
  bc
```

```
int p1 = p[ 1 ];
if( p[0] == 0 && p1 == 0 )

  lwz
  lwz
  cmp
  cmp
  crand
  bc      # 2-22 cycles
```

Avoiding aliasing

- May use the restrict keyword for similar pointer parameters
- Inlining improves visibility of expressions
- Aliasing manifests as
  - Seemingly redundant loads and stores
  - Bad scheduling
- Move loads to start of basic block and stores to the end

```
void Butterfly( float *p1, float *p2 )
{
  p1[ 0 ] = p2[ 0 ] + p2[ 1 ];
  p1[ 1 ] = p2[ 0 ] - p2[ 1 ]; // bad, p2[0] and p2[1] must be reloaded
}

void Butterfly( float *p1, float *p2 )
{
  float p20 = p2[ 0 ];
  float p21 = p2[ 1 ];
  p1[ 0 ] = p20 + p21;
  p1[ 1 ] = p20 - p21; // good, no need for reload
}
```

Type conversions and unions

- Ok to use unions on the stack frame

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

- Bad to use unions in classes – structure copies are ambiguous

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

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

PS3 intrinsics vs. inline assembler

- Intrinsics
  - schedulable
  - alias analysis
  - portable
    - atomic access
    - time base __mftb()
    - time-saving machine ops __fctiwz()
    - io __eieio()
    - debugging __builtin_frame_address()
    - system calls __system_call_nnn()

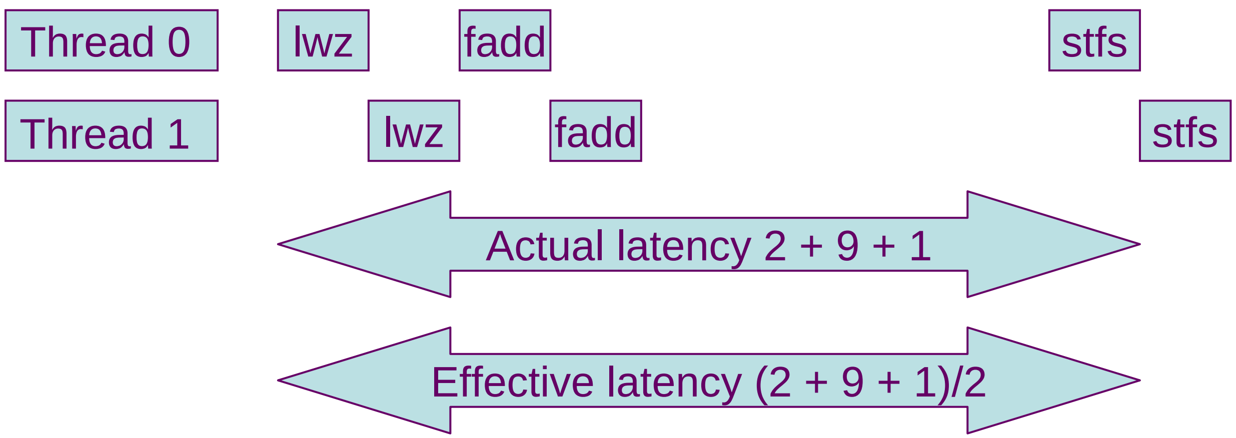
- Inline asm
  - machine specific
  - not schedulable
  - more flexible?

Vector classes dos and don'ts

- Use __attribute__((always_inline))
- Use access functions instead of unions
- Pass by value if and only if class has one data member
- Always use multiples of 16 bytes
- Mixing of float and VMX bad with GCC – use scalar class
- Do float -> integer conversions straight to memory
- Use "supervectors" to absorb latency
  - Groups of four or more vmx registers
  - Provides work to be done between register dependencies
  - Works over function calls

Multithreading effects on PS3

- Instructions are executed alternately: effective latency is halved
- Cache misses are covered by other thread
- Use SPUs for any available task
- Synchronization intrinsics have high latency
- Any second thread is better than the default.



Virtual function calls and switches

- Virtual function calls are incredibly useful
  – For high level control, AI and menus
- Virtual function calls are evil!
  – Very slow 50+ cycles
  – Only use for 100+ instruction functions
- Group values when using switches

```
switch( a ) // good: jump table
{
  case 1: ...
  case 2: ...
}
switch( a ) // bad: branch tree
{
  case 100: ...
  case 200: ...
}
```

  – Consider using look-up table before switch to cluster values

Console vs. PC programming

- PCs
  - have extra hardware to minimize the effects of latency
  - have fewer CPUs
  - cannot use precompiled display lists
  - designed to run legacy code
  - load from hard drives

- Consoles
  - are sensitive to latency
  - have many CPUs of different kinds
  - use precompiled display lists
  - run new code
  - load from DVD/Blu-ray

Console vs. PC programming

- Avoid using malloc - use pools instead and pre-allocate
- Pre-build display lists - CPU resource is precious, do not use it for rendering
- Design data structures to be spooled from DVD - do not use “serialize” methods or class factories
- Do not use global variables -global variable access is inefficient and uses data cache badly
- Avoid virtual functions / indirect calls
- Use fewer, larger modules for better interprocedural optimization – about ten to twenty modules is optimal for distributed builds.

Using SN Systems tools to examine your code

- Debugger
  - Pipeline analyzer
  - Randomly stopping execution can reveal hotspots

- Binary Utilities
  - Pipeline analyzer
  - Symbols

- Tuner
  - Look for hotspots – the instruction before the hotspot is the bad one!
  - PC sampling
  - Auto instrumentation of functions
  - User labels

Using SN Systems tools to examine your code

```
volatile char mem;
for( int j = 0; j != 10000; ++j )
{
    mem = ( mem - 1 >> 1 ) + 1;
    mem = ( mem - 2 >> 1 ) + 2;
}
```

Sample tuner PC sampling

```
     - -   f()
     - -   000103F0 38802710 li      r4,0x2710
     - -   000103F4 3C601001 lis     r3,0x1001           PIPE
     - -   000103F8 7C8903A6 mtspr   CTR,r4
  2029 -   000103FC 88834228 lbz     r4,0x4228(r3)     03 (000103F4) REG LSU
137785 -   00010400 7C840774 extsb   r4,r4             01 (000103FC) REG
  1095 -   00010404 3084FFFF addic   r4,r4,-0x1        01 (00010400) REG PIPE
  1739 -   00010408 7C840E70 srawi   r4,r4,1           01 (00010404) REG
   918 -   0001040C 30840001 addic   r4,r4,0x1         01 (00010408) REG PIPE
  3978 -   00010410 98834228 stb     r4,0x4228(r3)     03 (0001040C) REG LSU
   171 -   00010414 88834228 lbz     r4,0x4228(r3)     50 (00010410) PIPE LHS[01]
 61489 -   00010418 7C840774 extsb   r4,r4             01 (00010414) REG
  1325 -   0001041C 3084FFFE addic   r4,r4,-0x2        01 (00010418) REG PIPE
  1526 -   00010420 7C840E70 srawi   r4,r4,1           01 (0001041C) REG
  1182 -   00010424 30840002 addic   r4,r4,0x2         01 (00010420) REG PIPE
  3394 -   00010428 98834228 stb     r4,0x4228(r3)     03 (00010424) REG LSU
    21 -   0001042C 4200FFD0 bdnz    0x000103FC
     1 -   00010430 4E800020 blr
```

Instruction causing delay   delay   LHS within 1 cycle. (requires 20 for safety)

Most common hotspots:

- on branch targets (icache miss)
- on loads (dcache miss)
- on loads (Load-Hit-Store)

New SNC optimizations

- SSA analysis
  - Constant propagation
  - Memory optimizations
  - Use of VMX to replace int and float operations
  - Auto vectorization
  - Conversion of floating point compares to integer
  - Removal of fixed and zero iteration loops

SnMathLib

- Worked example of a complete math class for PSP® (PlayStation®Portable), PS3 and PC
- Shows correct construction of math libraries
- Scalar classes for mixed operation
- Includes “Supervector” class “quadquad” for better scheduling
  - Four vector operations of same kind at a time
  - Fills in gaps between instruction issues
- Extensive test suite
  - Performance
  - Accuracy (especially trig functions)

Essential reading

- Engineering a compiler (Cooper & Torczon)
- Wikipedia
  - http://en.wikipedia.org/wiki/Category:Compilers
- GCC internal documentation
  - http://gcc.gnu.org/onlinedocs/gccint/
- An interesting case study
  - http://www.flounder.com/optimization.htm