



Optimizing PC Games for 64-bit Windows® on AMD Athlon™ 64



In 5 easy steps

Mike Wall
Senior Member of Technical Staff
Advanced Micro Devices, Inc.

AMD64 Technology

- AMD Athlon™ 64

Single processor
Desktop, Mobile



- AMD Opteron™

Multi-processor
Workstation, Server



Windows® and AMD64 Technology

Unifying theme: Designed for Compatibility

- Processor: Native hardware support for 32-bit and 64-bit x86 code
- OS: 64-bit Windows® runs 32-bit and 64-bit applications side by side, seamlessly
- Source Code: A single C/C++ source code tree compiles to both 32-bit and 64-bit binaries

Step #1: get your code "64-bit clean"

- Build your project using the /Wp64 compiler switch
 - This switch is supported by 32-bit VS.NET compiler!
- Warns about non-portable code
- Fix it so your project compiles cleanly to 32-bit target
- Then you're ready to use the 64-bit tools

- Clean-up example 1: use new “polymorphic” data types
- Original code stores a pointer in a LONG

```
LONG userdata = (LONG) &max_coordinate;
```
- When built for a 64-bit target, this will truncate the pointer (64-bit value) by storing in a LONG (32-bit size).
- Use LONG_PTR instead:

```
LONG_PTR userdata = (LONG_PTR) &max_coordinate;
```
- Data type LONG_PTR is “a long the size of a pointer” so it grows to 64 bits when you compile for 64-bit target

- Clean-up example 2: a few API calls have been updated

- Old API call uses a 32-bit LONG for user data

```
LONG udat = myUserData;  
LONG v = SetWindowLong( hWnd, GWL_USERDATA, udat );
```

- New API call replaces the old one:

```
LONG_PTR udat = myUserData;  
LONG_PTR v = SetWindowLongPtr( hWnd, GWLP_USERDATA, udat );
```

- The old call is deprecated; the new call works for 32-bit and 64-bit targets, so you can (and should) change all your code

Step #2: get everything to build for 64-bit target

- There may be some additional compiler warnings/errors
- Data structure alignment is a common trouble spot
 - Data is “naturally aligned” in structs
 - Pointer members and polymorphic members grow
 - Shared data (e.g. access by assembly code) needs special care
- Data files shared between 32-bit and 64-bit processes may need special handling, especially if they contain pointers
- Skip assembly code porting, for initial 64-bit build

Step #3: benchmark and profile your 64-bit code

- Performance profile may differ from old 32-bit build
- There are new drivers...
- and there are new optimized libraries...
- and a new 64-bit optimizing compiler from Microsoft...
- all running on a new internal CPU micro-architecture.
- **Focus optimization efforts on real, measured hot spots!**
- You may be surprised by where your code spends time

Step 3: measure and analyze



Use AMD CodeAnalyst to profile your 64-bit code!!

- Timer-based & event-based profiler



Retire Cycle	Address	Disassembly	CodeBytes	Delta	Notes	Pipeline
3276	0x401430	movl pd xmm3, [\$+00032...]	66 0F 12 1D 68 2...	0		
3277	0x401438	movl pd xmm2, [\$+00032...]	66 0F 12 15 C0 2...	0		
3277	0x401440	lea eax, [rbp-02h]	8D 45 FE	0		
3277	0x401443	lea rcx, [rsp+02h]	48 8D 54 24 50	0		
3278	0x401448	lea rcx, [rsp+30h]	48 8D 4C 24 30	2		
3278	0x40144D	mov r8d, 0000003e8h	41 B8 E8 03 00 00	0		
3278	0x401453	movsd xmm0, xmm9	F2 41 0F 10 C1	5		
3315	0x401458	cvtsi2sd xmm1, ebp	F2 0F 2A CD	60		
3315	0x40145C	mulsd xmm0, xmm3	F2 0F 59 C3	0		
3320	0x401460	addsd xmm0, [\$+00032...]	F2 0F 58 05 70 2...	4		
3320	0x401468	mulsd xmm1, xmm3	F2 0F 59 CB	38		
3320	0x40146C	movsd [rsp+50h], xmm0	F2 0F 11 44 24 50	0		
3321	0x401472	movsd [rsp+58h], xmm0	F2 0F 11 44 24 58	1		
3321	0x401478	movsd [rsp+60h], xmm0	F2 0F 11 44 24 60	1		
3321	0x40147E	movsd [rsp+68h], xmm0	F2 0F 11 44 24 68	1		
3324	0x401484	cvtsi2sd xmm0, eax	F2 0F 2A C0	32	Bank conflict	
3324	0x401488	addsd xmm1, xmm2	F2 0F 58 CA	3		
3330	0x40148C	lea eax, [rbp-01h]	8D 45 FF	0		
3330	0x40148F	mulsd xmm0, xmm3	F2 0F 59 C3	71		



- You can get a detailed pipeline view of critical code sections
- Available free from AMD, go to amd.com developer section

Advanced Micro Devices - CodeAnalyst Performance Analyzer [C:\temp\float\float.caw]

float.caw

Session - Session1.sim

System Data System Graph float_test32.exe - Graph float

float_test32.exe - Session1.sim

Settings for float.caw

General Timer Config Event Config Simulation Config Symbols

AMD Optron Processor - Family 15 Model 5 Stepping 8

Counter 0 Counter 1 Counter 2 Counter 3

Event: 4b Dispatched prefetch instructions

Count: 1000

Src: C

Os User

Dispatched prefetch instructions

DCACHE accesses by locks (Revision C)

Apply OK Cancel

Samples Address Code Bytes Instruction

0x401198	CC		inc3
0x40119e	CC		inc3
0x40119f	CC		inc3
120.0x4011a0	40 8D C1		mov rcx, cx
65.0x4011a3	0F 57 C0		xorps xmm0, xmm0
0x4011a6	F3 0F 10 00		movss xmm0, [rax]
66.0x4011ea	F3 0F 59 C1		mulss xmm0, xmm1
100.0x4011ee	F3 0F 11 00		movss [rax], xmm0
164.0x4011b2	C3		retmq
0x4011b3	55		push rbp
0x4011b4	53		push rbx
0x4011b5	51		push rcx

6 instructions, Total: 515 samples, 69.88% of samples in module float_test64.exe, 12.92% of tot

Step #4: tune for maximum 64-bit performance

- In 64-bit mode, AMD64 technology offers 8 extra GPRs and 8 extra SSE registers: this is twice what 32-bit x86 supports!
- Many of the old C/C++ tricks still work, some extra-well
 - The compiler optimizes, but you can help it at source level
- Compiler intrinsic functions for SSE, SSE2 and other funcs
 - Portable across 32-bit and 64-bit targets, compiler does reg allocation
- Assembly code can still be used for absolute max performance
 - Vectorize your 64-bit SSE/SSE2 code, tweak instruction scheduling, etc.

- Use the 64-bit compiler's optimization features
- Good ol' standard optimization switches
 - Compile with `/O2` or `/O2b2`, and use `/fp:fast`
- Whole Program Optimization (WPO)
 - Compile with `/GL` and link with `/LTCG`
 - Enables more function inlining, other cross-module improvements
- Profile Guided Optimization (PGO or "Pogo")
 - Build instrumented binaries, run your workload, re-link
 - Can improve function layout (I-cache usage) and branch flow

- Source-level optimization example 1: loop unrolling
- The compiler unrolls loops automatically, but manual unrolling can help greatly if you introduce *explicit parallelism*

```
double a[100], sum;
sum = 0.0;
for (int i = 0; i < 100; i++) {
    sum += a[i];    // no parallelism possible
}
```

- The compiler must use a single accumulator “sum” and perform addition in-order, creating a long dependency chain and leaving execution units idle much of the time
- How can this be improved?

Step 4: optimize



- Manually unroll the loop into *parallel dependency chains*

```
double a[100], sum, sum1, sum2, sum3, sum4;
sum1 = sum2 = sum3 = sum4 = 0.0;
for (int i = 0; i < 100; i += 4) {
    sum1 += a[i];           // these four
    sum2 += a[i+1];       // chains can
    sum3 += a[i+2];       // run in
    sum4 += a[i+3];       // parallel
}
sum = sum1 + sum2 + sum3 + sum4;
```

The switch
/fp:fast
can do this
automatically
in many cases.

Use it!

- With 4 separate dependency chains the execution units can be kept busy with pipelined operations... over 3x faster here!
- This trick works especially well in more complex loops with AMD64 technology because of all the registers available.

- Source-level optimization example 2: aliasing
- Perhaps the biggest optimization roadblock for the compiler is *aliasing*. Pointers may step on each others' data

```
int *a, *b, *c;
for (int i = 0; i < 100; i++) {
    *a += *b++ - *c++; // b or c may point to a
}
```

- The compiler must be cautious, and write to memory for each loop iteration. It cannot safely keep the sum in a register.
- How can this be improved?

- Use the `__restrict` keyword to help the compiler
- Apply your external knowledge that `*a` does not alias `*b` or `*c`

```
int* __restrict a;  
int *b, *c;  
for (int i = 0; i < 100; i++) {  
    *a += *b++ - *c++; // no aliases exist  
}
```

- Now the compiler can safely keep the sum in a register, and avoid many memory writes.
- Read more about keyword `__restrict`, `declspec(restrict)` and `declspec(noalias)` in Microsoft docs. They are powerful.

- Source-level optimization example 3: struct data packing
- Structure data members are “naturally aligned”
- Padding may be added when you compile for 64-bit

```
struct foo_original { int a, void *b, int c };
```
- 12 bytes in 32-bit mode, but 24 bytes in 64-bit mode!
- Fix it by re-ordering elements for better packing

```
struct foo_new      { void *b, int a, int b };
```
- 12 bytes in 32-bit mode, 16 bytes in 64-bit mode
- Also re-order struct elements for better cache locality

- Compiler intrinsic examples: SSE and SSE2

```
__m128  _mm_mul_ss( __m128 a, __m128 b );
```

SSE MULSS scalar single-precision multiply instruction

```
__m128d  _mm_add_pd( __m128d a, __m128d b );
```

SSE2 ADDPD packed double-precision add instruction

```
__m128i  _mm_load_si128( __m128i *p );
```

SSE2 MOVDQA instruction for 128-bit integer

```
__m128d  _mm_set_pd( double x, double y );
```

SSE2 initialize a packed vector variable

```
__m128d  _mm_setzero_pd( );
```

SSE2 XORPD initialize packed double to zero

```
__m128i  _mm_cvtsi32_si128( int a );
```

SSE2 MOVD load a 32-bit int as lower bits of SSE2 reg

- Compiler intrinsic examples: other goodies

```
void __cpuid( int* CPUInfo, int InfoType );
```

for detecting CPU features

```
unsigned __int64 __rdtsc( void );
```

for reading the timestamp counter (cycle counter) and accurately measuring performance of critical code sections

```
__int64 __mul128( __int64 Multiplier, __int64 Multiplicand,  
                 __int64 *HighProduct );
```

fast multiply of two 64-bit integers, returning full 128-bit result (lower 64 bits are return value, upper 64 bits by indirect pointer)

```
__int64 __mulh( __int64 a, __int64 b );
```

fast multiply of two 64-bit integers, returning the high 64 bits

```
void __mm_prefetch( char* p, int i );
```

software prefetch instruction for reading data into CPU cache

```
void __mm_stream_ps( float* p, __m128 a );
```



streaming store of data to memory, bypassing CPU cache

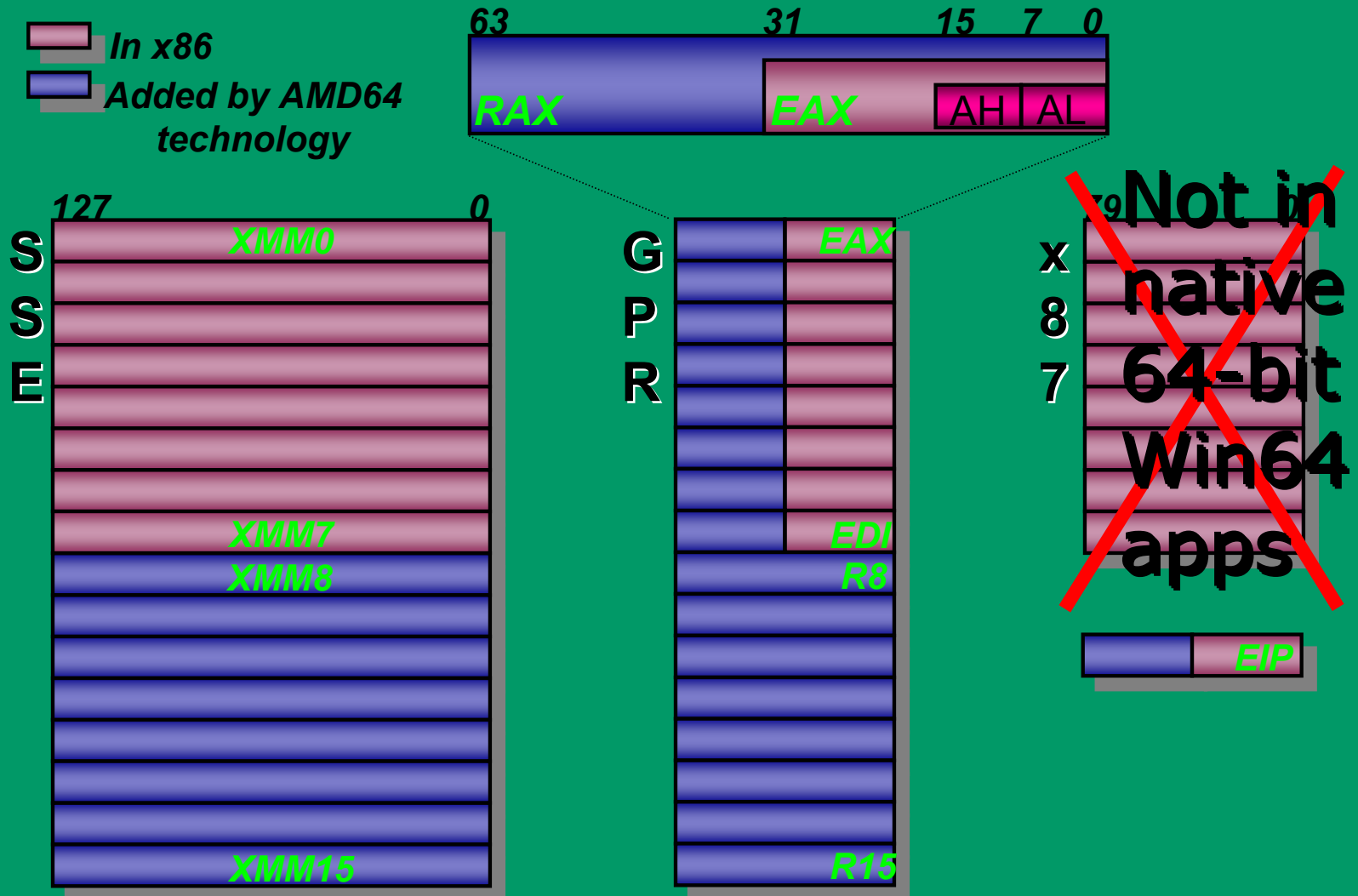
- Assembly code is still worthwhile for maximum performance in certain critical inner loops
- In-line `_asm` code is not supported for 64-bit code, use MASM
- Pay attention to prolog/epilog, it's different... and faster
 - Values passed in registers, rarely pushed on stack
 - Certain regs are volatile, others are non-volatile
- Be careful about data layout: 64-bit code may be different

Step 4: optimize



AMD64 Programmer's Model

 In x86
 Added by AMD64 technology



Step 4: optimize



- Assembly code example: software pipelining
- This is the asm version of the parallel dependency chain hack

```
movapd    xmm2, xmm0;
mulpd     xmm2, xmm1;    c = z_real x z_imag
mulpd     xmm0, xmm0;    a = z_real x z_real
mulpd     xmm1, xmm1;    b = z_imag x z_imag
subpd     xmm0, xmm1;    z_real = a - b
addpd     xmm2, xmm2;    c = c x 2
addpd     xmm0, xmm12;   z_real = z_real + c_real
movapd    xmm1, xmm2;
addpd     xmm1, xmm13;   z_imag = c + c_imag
```

- Lots of data dependencies limiting performance... what to do?

Step 4: optimize



- Use the extra registers and implement *software pipelining*

```
movapd    xmm2, xmm0;
movapd    xmm6, xmm4;
mulpd     xmm2, xmm1;    c = z_real x z_imag
mulpd     xmm6, xmm5;
mulpd     xmm0, xmm0;    a = z_real x z_real
mulpd     xmm4, xmm4;
mulpd     xmm1, xmm1;    b = z_imag x z_imag
mulpd     xmm5, xmm5;
subpd     xmm0, xmm1;    z_real = a - b
subpd     xmm4, xmm5;
addpd     xmm2, xmm2;    c = c x 2
addpd     xmm6, xmm6;
addpd     xmm0, xmm12;   z_real = z_real + c_real
addpd     xmm4, xmm14;
movapd    xmm1, xmm2;
movapd    xmm5, xmm6;
addpd     xmm1, xmm13;   z_imag = c + c_imag
addpd     xmm5, xmm15;
```

No need to overlay the two chains quite this tightly.

The CPU re-orders instructions aggressively, so dependency chains only need to be reasonably close together.


For max performance...
experiment!

- Working 2 independent data sets improves performance >35% here!

Step 5: beer



- After you have your 64-bit code running like blazes...
- Have a beer



32-bit and 64-bit
optimization and graphics demo

- Inserting a second dependency chain in the loop dramatically improves performance on both 32-bit and 64-bit
- 64-bit code benefits more, because of the extra SSE regs
- 64-bit mode provides a 30%+ boost over 32-bit mode

	32-bit	64-bit	
1 chain	102	112	~10% 64-bit benefit
2 chains	147	192	~30% 64-bit benefit
Gain =	44%	71%	

- Compile with `/Wp64` all the time, and `/O2b2 /fp:fast` for 64-bit
- Go to www.amd.com and get all the AMD docs
 - “Develop with AMD” and “AMD64 Developer Resource Kit”
 - Download and use the CodeAnalyst profiler, for 32 and 64-bit code
 - Learn how to use the 64-bit PSDK compiler with Visual Studio
 - AMD Developer Center in Sunnyvale! Visit us, or remote access
- Go to Microsoft BetaPlace, get all the AMD64 technology stuff
 - Windows 64-bit OS and Platform SDK on Betaplace (build 1159)
 - Current AMD64 Platform SDK based on VC6.0 libs
 - For ATL/MFC, CRT, STL 7.1 lib files: e-mail **libs7164@microsoft.com**
 - DirectX for AMD64: e-mail **DX9Beta@microsoft.com**
- Go to MSDN and Microsoft.com for more docs
 - Search for 64-bit, AMD64, or “64-bit Extended”
 - Read about new 64-bit compiler features, intrinsics, etc.

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft and Windows are registered trademarks of Microsoft Corporation. Other product names used in this presentation are for identification purposes only and may be trademarks of their respective companies.