Porting and Optimizing Applications on 64-bit Windows® for AMD64 Architecture

July 21, 2004

Abstract

The new 64-bit Windows® operating systems include full 64-bit support for AMD Athlon™ 64 and AMD Opteron™ processors. The new OS takes advantage of new architectural features of these AMD64 processors, to offer improved performance, scalability and security. While the 64-bit OS offers excellent backward compatibility and performance with 32-bit applications, developers can often achieve substantial performance benefits by porting code to native 64-bit mode. Some common portability issues are addressed in this paper, and a variety of optimization techniques for the 64-bit architecture are described so developers can maximize performance when they port their applications.

Contents

Introduction 3
Porting Your Apps to 64 Bits: Do It When You're Ready 3
I Was Just Kidding, You Really Should Port Now! 3
64-bit Code is Just Way Better Than 32-bit Code 4
90% of Porting is Just Recompiling 4
Sharing Data Can Be a Bit Tricky 5
Boosting Performance Even More When Porting 6
Compiler switches for optimization 6
Memory mapped files 7
Compiler intrinsic functions 7
Controlling data alignment 8
Enabling more aggressive register usage 8
C library functions and other optimized libraries 8
Controlling thread and memory affinity 9
Assembly code optimizations 10
Big Number Math 10
Software pipelining 11
The CodeAnalyst Profiling Tool 12
Summary and Addendum 13
Resources and References 14

Windows Hardware Engineering Conference
Author's Disclaimer and Copyright:

© 2004 Advanced Micro Devices, Inc.
All rights reserved.

AMD, AMD Athlon and AMD Opteron are trademarks of Advanced Micro Devices, Inc.

WinHEC Sponsors' Disclaimer: The contents of this document have not been authored or confirmed by Microsoft or the WinHEC conference co-sponsors (hereinafter "WinHEC Sponsors"). Accordingly, the information contained in this document does not necessarily represent the views of the WinHEC Sponsors and the WinHEC Sponsors cannot make any representation concerning its accuracy. THE WinHEC SPONSORS MAKE NO WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS INFORMATION.

Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

Introduction

The new 64-bit Windows® operating system includes full 64-bit support for AMD Athlon™ 64 and AMD Opteron™ processors. The new OS takes advantage of new architectural features of these AMD64 processors, to offer improved performance, scalability and security.

Significantly, these 64-bit features are enabled while maintaining excellent compatibility and performance for existing 32-bit Windows® applications. The AMD64 technology is implemented as the natural extension and evolution of the established, industry standard 32-bit x86 technology.

Porting Your Apps to 64 Bits: Do It When You're Ready

The 64-bit Windows® operating system has vastly more resources available, both for the OS and for applications. In fact, the 64-bit OS runs many 32-bit applications at performance levels equal to, or even greater than, the 32-bit OS. This is a compelling benefit for end users. So the first scenario of interest for any software developer is verifying 32-on-64 validation of their existing 32-bit application.

Under 64-bit Windows®, each 32-bit application is free to access a private, exclusive virtual memory space up to 4GB in size. This is a major improvement over the 32-bit OS, where applications are limited to 2GB virtual address space (or at best, a 3GB space) and less physical memory is typically available. Access to the larger memory space can offer substantial performance gains, an immediate benefit for early adopters running 32-bit applications on the 64-bit OS.

Furthermore, the 64-bit OS has far more memory space for System Cache, data tables, buffers and other resources, so just about everything from disk I/O to task switching runs faster. 64-bit Windows® is quite simply "a better Windows."

I Was Just Kidding, You Really Should Port Now!

Even with the additional capabilities of the 64-bit OS, a 32-bit application probably cannot tap the full performance potential of the 64-bit Windows® platform and AMD64 technology. Any 32-bit application that is compute-intensive or memory-intensive can probably get a performance boost from porting to native 64-bit mode.

Any application that can benefit from higher performance is an excellent candidate for porting. This includes 3D rendering and animation, games, simulation, video editing, streaming media and codecs, compilers and programming tools, databases, web servers, transaction processors and most server applications. And as users adopt the 64-bit OS, they will seek out 64-bit versions of their primary applications.

One other good reason to port your code: in some cases, 64-bit code is required for compatibility. DLLs or other "in process" plug-ins need to be 64-bit code to run in a 64-bit process. A 32-bit application and a 64-bit application can run side-by-side on the same machine, but one application cannot mix 32-bit and 64-bit code in the same process. Each application must be pure 32-bit, or pure 64-bit. So if you write static library code, DLLs, plug-ins or other shared code that will be used in various applications, you must port it to 64-bit mode for compatibility with the 64-bit applications that will use it.

In a similar vein, device drivers that run on the 64-bit OS must all be native 64-bit code. This is an OS platform requirement; 32-bit device drivers will not work on the 64-bit OS.

64-bit Code is Just Way Better Than 32-bit Code

Porting your application to 64-bit Windows® on AMD64 technology can pay off in many ways. The AMD64 architecture offers more than just a greatly expanded memory address space. It also fixes the most serious limitation of the venerable x86 instruction set: the limited number of registers. The 32-bit x86 supports only 8 general purpose registers, several of which are already consumed by essential functions like the stack pointer. The AMD64 architecture defines 8 additional general purpose registers, bringing the total to 16. All 16 GPRs are 64 bits in size.

Furthermore, in the AMD64 architecture the number of SSE/SSE2 registers is doubled from 8 to 16 as well. The full SSE and SSE2 instruction sets are supported, and can operate on all 16 of these registers.

The Microsoft 64-bit compiler makes excellent use of all the registers. Temporary variables are stored in registers and rarely need to be stored and retrieved from memory. Parameters (up to 4) are passed to functions in registers, rather than in stack memory. Floating point code is compiled as SSE/SSE2 instructions with full access to the 16 SSE registers.

The bottom line: all the additional registers mean higher performance for most code, especially compute-intensive code. For example, computer graphics and simulation codes typically process 3D vertex data by performing matrix math. Matrices are usually 3x3 or 3x4 elements. With only 8 SSE registers available, these matrices must be stored in memory and repeatedly reloaded into registers. With 16 SSE registers the matrix can be kept entirely in registers.

Codecs for audio and video data can also benefit from the additional registers. With vectorized code, you can store 4 floating point values or 8 16-bit integers in an SSE register. This means 64 floating point values (4 x 16) or 128 integers (8 x 16) can be stored in the SSE registers! A large amount of processing can proceed entirely within registers; results can be streamed out to memory only upon completion.

90% of Porting is Just Recompiling

The 64-bit Windows® APIs remain virtually identical to the 32-bit Windows® APIs. Only a few data types have been changed, to reflect certain items that grow to 64-bit data size. So for most applications, porting boils down to making sure you've got all the source code you need (or 64-bit binaries for third-party libraries and DLLs you use). Then you recompile the code with the 64-bit Microsoft compiler.

But even before attempting to recompile, you can easily check for portability problems. Microsoft has provided a special switch in their 32-bit compiler for this purpose. Build your project with the /Wp64 switch, and the compiler will emit warnings about non-portable sections of code. Using the /Wp64 switch is highly recommended, and it's an efficient way to identify and fix portability problems, and in the process you will even make your 32-bit code cleaner and more readable.

A typical compile-time problem involves unsafe data type conversions. For example, certain Windows® programs might do something like this:

```
LONG  userdata = (LONG) &max_coordinate;
```

This works for 32-bit Windows® because a LONG and a pointer are both 32 bits. But in 64-bit Windows®, pointers grow to 64 bits while LONG remains 32 bits. So if you compile for 64-bit Windows®, this code truncates the 64-bit pointer which is probably not the intended behavior. This kind of "Classic Bad Programming" will be flagged by the /Wp64 compile flag, and can be cleaned up before you even start using the 64-bit tools. A handful of new data types will assist you in this cleanup, as described below.

Once you start building with the 64-bit compiler, most code should build without problems. In 64-bit Windows®, only pointers (and variables that store pointers) grow to 64 bits. INT, LONG, and other data types remain the same size. Pointers are automatically promoted to 64-bit size. Data structures that include pointers will grow, but this should not cause errors if structure members are indexed by name and no implicit assumptions were made about structure packing and layout.

There will almost always be a few compile errors. Here is one common compile error seen when moving to 64-bit Windows®.

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

Why is this a compile error? Because "udat" is a 32-bit LONG, but 64-bit Windows® must provide support for 64-bit values in this function. Therefore, this is among the few Windows® API functions that has been tweaked for the 64-bit OS. However, the tweak is minor and it compiles cleanly for both 32-bit and 64-bit Windows®. The new function is SetWindowLongPtr, and is used like this:

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

You see the new data type LONG_PTR used above. LONG_PTR is a "polymorphic" data type. A LONG_PTR is 32 bits (i.e. the same as a LONG) when compiled for 32-bit Windows®, and 64 bits when compiled for 64-bit Windows®. LONG_PTR means "a long that is the same size as a pointer."

SetWindowLong and the sister function GetWindowLong have been deprecated, replaced by SetWindowLongPtr and GetWindowLongPtr. These are likely to be needed in most Windows® applications. All the required code tweaks are straightforward and similar to the above example, and compile for both 32-bit and 64-bit environments. These functions are documented in detail on MSDN.

There may be other places in your code where certain LONG, ULONG or INT variables must be changed to polymorphic types for cross-platform compatibility. For example, be aware that LPARAM, WPARAM, LRESULT, HWND and size_t scale up to 64 bits under 64-bit Windows®. Many apps blindly convert size_t data to an INT, which will not work properly in 64-bit mode.

Sharing Data Can Be a Bit Tricky

64-bit code will typically have different alignment for data structures, because certain elements will be promoted to 64-bit data size. This is not a problem within an application; if the code is written cleanly, the compiler will "do the right thing."

Care must be taken when assembly language code shares data with a C/C++ program, but since the ASM code is 64-bit specific, there are no cross-platform compatibility requirements here. (see MSDN or SDK documentation for the new register-based AMD64 architecture calling convention when writing assembly code)

However, data size/alignment differences between 32-bit and 64-bit applications can cause compatibility problems if data is blindly imported or exported. For example, certain data files might be shared between 32-bit and 64-bit applications, and files may contain pointer-size data which must be handled a bit differently by the 64-bit code. Or certain data structures may be shared in memory, or shared using Interprocess Communication (IPC) methods like DDE or COM, and the structure layout must be invariant to ensure compatibility between 32-bit and 64-bit applications.

Windows® offers some fixed-size data types to facilitate this kind of data sharing. For example, __int64 is an INT that is always 64 bits in size, whether the application is compiled for 32-bit Windows® or 64-bit Windows®. You might use fixed-size types in defining structures that describe file headers, for example.

For a detailed list of new data types in 64-bit Windows®, search the MSDN web site. It's also instructive to look at the basetsd.h file where data types are defined.

Boosting Performance Even More When Porting

Many performance benefits come automatically when applications are ported to 64-bit Windows® for AMD64 technology. For example, all the benefits of the extra general purpose registers and SSE registers, the vastly larger address space, and faster OS calls are inherently realized when applications are ported. But with a little extra care and attention, developers can get additional performance benefits when they port. MSDN and the SDK have detailed documentation on most of the following features:

Compiler switches for optimization

The Microsoft compiler for the AMD64 technology supports some excellent optimization capabilities.

The basic optimization switch **/O2** enables numerous code quality improvements for speed, and it should always be evaluated. This switch is quite safe, and it's almost guaranteed to improve performance. Switch **/fp:fast** should also be used for most code; it enables aggressive floating point performance optimizations.

Compiler switch **/GL** and linker switch **/LTCG** provide link-time code optimizations, and these switches are used for advanced Whole Program Optimization (WPO) and Profile Guided Optimization (PGO). Whole Program Optimization enables code improvements that span across functions and modules. Profile Guided Optimization actually instruments the executable code and gathers run-time statistics, then it uses those statistics and modifies the code for improved performance when it is re-linked. Using PGO takes a little more time and effort than most compiler features, but the performance benefits can be worthwhile.

Compiling with the **/GS** switch builds code with enhanced security against certain types of viruses, and this switch should always be used.

See the compiler documentation for full details on how to use all of the switches.

Memory mapped files

Windows® has long supported the concept of memory mapped files, but this feature really becomes compelling in the 64-bit execution environment. A set of functions including CreateFileMapping and MapViewOfFile allows an application to treat a file just like a chunk of memory. The file is read and written using pointers or array notation, just like memory.

The application code is greatly simplified, because it does not have to deal with explicit file I/O commands or memory buffer management. Furthermore, the disk I/O operations can be faster because the OS can buffer large amounts of the data in memory. This all works very well under 64-bit Windows® because the vast virtual address space permits files of essentially any size to be mapped and accessed in this manner.

Compiler intrinsic functions

In-line assembly code is not supported in the 64-bit Microsoft compiler (assembly code must be implemented in a separate MASM file, assembled, and linked with the app). However, many architectural features of the AMD64 platform can be directly accessed in C/C++ source code, using special compiler intrinsic functions that are supported by the Microsoft 64-bit compiler. This includes many operations that otherwise would require assembly language.

MSDN has full specs for compiler intrinsic functions, but a few key examples are:

```
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 the full 128-bit result
(lower 64 bits are the 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 L1 cache

```
void _mm_stream_ps( float* p, __m128 a );
```
streaming store of data to memory, bypassing CPU cache

Furthermore, virtually all SSE and SSE2 instructions are implemented as compiler intrinsics. Fully vectorized floating point or integer code can now be written at the source level using these intrinsic functions. The SSE and SSE2 intrinsics are compiled with full optimization to use all 16 SSE registers, and are fully cross-compatible with 32-bit compilers. The 32-bit and 64-bit compilers take care of optimized instruction scheduling and register allocation.

Note: SSE and SSE2 supercede the older MMX, 3D Now! and x87 instruction extensions for all native 64-bit Windows® applications.

Controlling data alignment

By default, the 64-bit Microsoft compiler aligns data in memory for excellent performance. However there are certain cases where explicit alignment control can improve cache behavior, and also some kinds of data that favor or require more strict alignment. For example, SSE vector data must be 16-byte aligned. The 64-bit compiler provides the __declspec(align(#)) keyword for explicit alignment control, and it is used like this:

```
//  16-byte aligned data
__declspec(align(16))  double init_val [ 3.14, 3.14 ];

//  SSE2 movapd instruction
_m128d  vector_var  = _mm_load_pd(init_val);
```

Enabling more aggressive register usage

One of the hardest challenges for a compiler is aliasing. When code is reading and writing memory, it is often impossible (at compile time) to determine if more than one pointer is actually accessing the same address, i.e. more than one pointer can be an “alias” for the same data. Therefore in a loop that is both writing and reading memory, the compiler must often be very conservative about what data is kept in registers. This conservative use of registers can make code perform sub-optimally.

The keyword __restrict can be used to alleviate this problem. The __restrict modifier can be applied to a pointer in a function declaration or function body, and it tells the compiler that nothing “sneaky” will happen to the data that is accessed by this pointer. For example:

```
void mul_arr (double* __restrict X, double* __restrict Y);
```

Now the compiler can safely stash any data from the X and Y arrays in registers.

Two related examples: if a function returns a pointer type, you can declare that the memory pointed to by that return value is not aliased, like this:

```
__declspec(restrict) double* v_dot (float* v1, float* v2);
```

Finally, you can also tell the compiler that a function does not interfere with the global state except through pointers in its parameter list, like this:

```
__declspec(noalias) double* v_dot (float* v1, float* v2);
```

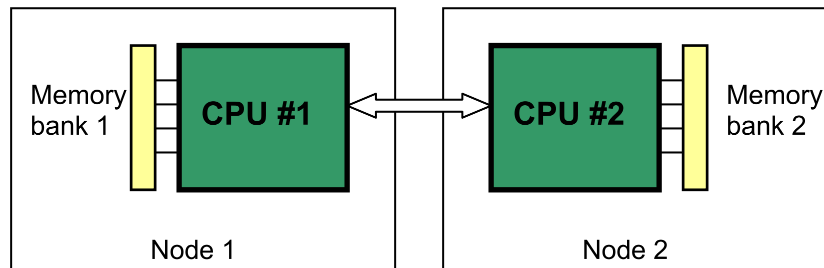
C library functions and other optimized libraries

Most of the common libc functions are highly optimized in the 64-bit Microsoft compiler, including memcpy, memmove, memset, strlen, strcpy, strcat, and most of the other common string functions. These functions are already used in many C/C++ applications, and they should continue to be used; any effort to “reinvent the wheel” with these basic functions will most likely degrade performance.

Additional highly optimized code libraries are available for the AMD64 platform. For example, optimized BLAS (Basic Linear Algebra Subprograms), LAPACK (Linear Algebra PACKage), FFT (Fast Fourier Transform) and other routines are available free in AMD’s ACML library. Optimized DirectX graphics code is included in Microsoft DirectX. New optimized libraries are being developed; search the Web or ask your software suppliers. Using optimized library code is one of the fastest and easiest ways to improve the performance of an application.

Controlling thread and memory affinity

Multiprocessor AMD64 platforms running 64-bit Windows® offer new avenues for application optimization. The AMD64 chip architecture includes an integrated memory controller on every processor. System memory is physically organized as banks, typically one bank attached to each processor or "node" in the system. This arrangement is generally transparent to software; all system memory is accessible to all processors.



This system architecture is called ccNUMA (Cache Coherent Non-Uniform Memory Architecture). The 64-bit Windows® OS is aware of the ccNUMA architecture, and it automatically allocates memory and threads accordingly for good performance. However, certain multi-threaded applications may benefit by exercising manual control of thread and memory affinity.

A thread can manually set the processor affinity, so it preferentially executes on a particular processor or set of processors.

```
DWORD_PTR SetThreadAffinityMask(
  HANDLE hThread,
  DWORD_PTR dwThreadAffinityMask
);
```

For example, an application may operate on largely separate data in each execution thread. To ensure that this data is placed in memory banks that are most local, memory blocks should be obtained by calling VirtualAlloc from within the thread which will use that memory block, once that thread is running on the preferred node. Memory should be initialized, and thus committed to local physical RAM on that node, by choosing the MEM_COMMIT allocation type when calling VirtualAlloc. See MSDN documentation for details.

In certain applications, it may even be optimal to "clone" blocks of read-only data that are heavily accessed by multiple threads on different processors. For example if the system is configured with large amounts of physical RAM, and the application is not performance-limited by total available RAM, there may be a performance advantage in "wasting" memory by making local copies of frequently accessed data. Performance gains would depend very much on the particular data access patterns of the application, and on the system's memory configuration, but this is one more trick to keep in your optimization bag.

See MSDN for more details on 64-bit Windows® ccNUMA OS features, and the related API functions including SetThreadAffinityMask, GetNumaProcessorNode, GetNumaHighestNodeNumber, GetNumaAvailableMemoryNode and others.

Assembly code optimizations

Historically, developers have used assembly language optimization to get maximum performance for certain critical inner loops. Compilers have steadily been improving, but there is still room for assembly code optimization, and the 64-bit Microsoft compiler and MASM 64-bit assembler provide this capability.

However, before taking the plunge and converting 32-bit assembly code to 64-bit assembly code, it is recommended that an equivalent C/C++ code path (if one exists) be compiled with the 64-bit compiler, and benchmarked. Because 32-bit assembly code was often written long ago, and tweaked for older CPU microarchitectures, modern 64-bit compiled code may actually run faster than the old ASM code! Test and verify that assembly code optimization is still appropriate before moving ahead.

Assembly programmers on the AMD64 platform will be productive quickly: the instruction set is essentially the same as the old x86, plus 64-bit addressing and extra registers. For example, a section of 32-bit assembly code might look like this:

```
add    eax,  DWORD PTR [esi + ecx*8]; adjust offset
movdqu xmm0, XMMWORD PTR [edi + eax]; load data from memory
```

And in the 64-bit world, it might look like this:

```
add    rax,   DWORD PTR [rsi + rcx*8]; adjust offset
movdqu xmm12, XMMWORD PTR [rdi + rax]; load data from memory
```

There are a few things to notice here. You can see the instructions, addressing modes, and overall format are the same. In the 64-bit assembly code, the names of the “legacy” registers (eax, ecx, edx, esi …) remain the same when used as 32-bit values, but they begin with an “r” when used as 64-bit registers (rax, rcx, rdx, rsi…). The 8 new general purpose registers are named r8-r15.

All 16 of the GPRs can be used as 8, 16, 32 or 64 bits. One subtle but useful aspect of the AMD64 instruction set: when a GPR is updated as a 32-bit value, the upper 32 bits are cleared to zero. This is typically the desired behavior, and it executes faster than preserving the upper 32 bits.

Big Number Math

64-bit integers can be added or subtracted, and the MUL and IMUL instructions can operate on two 64-bit values, to produce a full 128-bit result:

```
mov    rax, [rsp + var1];  load first 64-bit operand
add    rax, r12;           add a 64-bit value
mul    rcx;   now RDX:RAX contains 128-bit product rax*rcx
```

This type of “big number math” is only available to 64-bit code on the AMD64 platform, and it can greatly improve performance. For example, it accelerates core RSA encryption algorithms by over 400% compared to 32-bit x86 code.

Assembly code can also be used to implement optimized vectorization of floating point or integer calculations for processing audio and video. Codecs such as MP3 audio, MPEG-2 video and DivX video can run faster when vectorized algorithms are implemented using all 16 SSE registers provided by the AMD64 instruction architecture. White papers and presentations exist on this subject.

Software pipelining

The performance of certain critical loops is limited by data dependency chains. Execution units cannot be well utilized because they are waiting for results from previous operations. The extra registers available in 64-bit mode can help here, by enabling an optimization technique called Software Pipelining. The basic idea is to process a second, separate dependency chain in parallel. The two chains are interwoven, so the execution units can generally stay busy by alternating between the two chains.

For example, the inner loop of a Mandelbrot Set calculator might look like this

```
    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
```

There are dependency chains in this loop, which limit its performance. Extra registers can be used to software pipeline the loop, for example:

```
    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;
```

In this code sequence, software pipelining improves performance by 35%. (note: the second dependency chain is shown interleaved instruction-by-instruction, but this is not generally necessary; it only needs to be somewhere close to the first chain, and the CPU can execute it out-of-order in parallel with the first chain.)

Remember, 64-bit assembly code for Windows® cannot use the older MMX, 3D Now! and x87 instruction extensions; they have been superseded by SSE/SSE2.

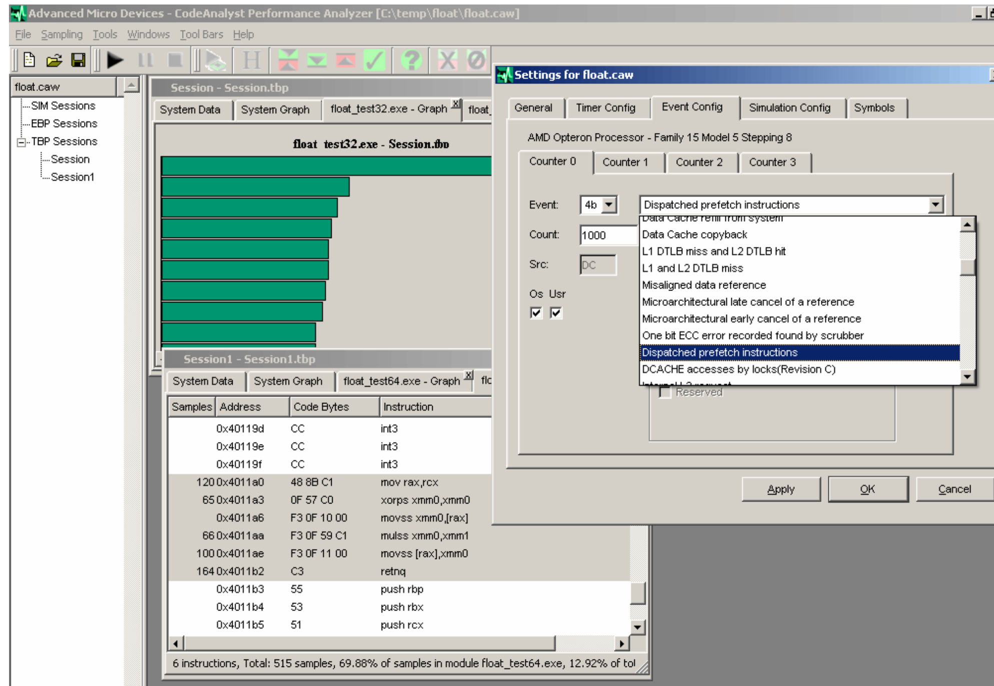
Certain registers are "volatile" and others are "non-volatile." If you use non-volatile registers in your assembly code, you must be sure to save the registers in accordance with the AMD64 ABI. Please refer to MASM & ABI documents (in PSDK/DDK and on MSDN) for more information regarding helper macros for emitting unwind information, stack format, parameter passing and register usage.

For more details on the AMD64 architecture and instruction set, please see the AMD64 Architecture Programmers' Manuals. For detailed instruction timings and much more about optimization techniques, see the Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors. These are all on the AMD web site.

The CodeAnalyst Profiling Tool

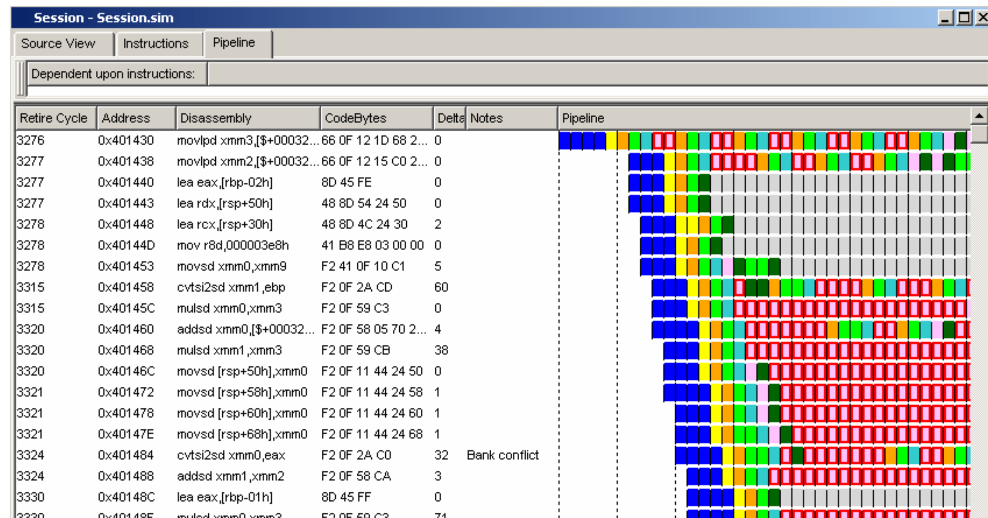
CodeAnalyst is an easy-to-use profiling tool available free from AMD. Just a single session with CodeAnalyst can provide valuable and often surprising insight into the performance profile of an application, and enable well-targeted optimization efforts.

CodeAnalyst can perform simple timer-based profiling. It can also trigger on and count a multitude of different events, such as misaligned data reads, cache misses, branch mispredicts, floating point exceptions, and many others.



typical CodeAnalyst profiling session

As an additional window into the internal performance details of your application, CodeAnalyst also provides a pipeline simulation view with cycle-by-cycle display of instruction decoding and processing in the various execution units.



Summary and Addendum

Porting Windows® applications to 64-bit Windows® for AMD64 technology delivers numerous benefits for performance. This platform offers smooth migration from the ubiquitous 32-bit world to a compatible, high-performance 64-bit world.

Addendum

64-bit Windows: Virtual Memory how it was always meant to be!

In the pioneering days of computer science, the concept of Virtual Memory was a breakthrough. Virtual Memory is an abstraction that allows software to access all data in a uniform manner, independent of the particular physical machine it's running on. This abstraction greatly simplified software design and enabled rapid innovation at the hardware level, without ever sacrificing compatibility.

In order for Virtual Memory to function correctly, the virtual address space *must* be much larger than the physical address space. The OS must be able to hand out large chunks of virtual memory on demand, never running out of address space, and never needing to reclaim or reorganize the virtual memory space. This assumption has worked very well. Until now.

In the 32-bit world, this basic assumption of Virtual Memory is no longer true. The relentless march of Moore's Law has propelled the physical memory size to approach the virtual memory size: 4 gigabytes (or even less in practice). As a result, the OS can no longer guarantee that virtual memory blocks are available to applications. As applications run, the virtual memory space becomes fragmented. Even if some portion of physical memory is available for use, there may not be a contiguous block of virtual address space available, so that physical memory becomes effectively not usable. Power users can see their machine slow down, as large applications eventually run out of memory.

A 64-bit OS like 64-bit Windows implements Virtual Memory the way it was always meant to be: a virtual address space that is vastly larger than the physical memory. Now all the physical memory can be put to good use. 4GB… 16GB… and beyond.

About the author

Mike Wall is a Senior Member of Technical Staff at Advanced Micro Devices, Inc. He has delivered numerous presentations on software optimization and AMD microprocessor architecture, and developed some of the optimization techniques in the AMD64 Optimization Guide. In 2003 he worked with the Microsoft compiler team to improve 64-bit compiler code quality, and wrote optimized library routines for the 64-bit Windows tools. His charter is to enable Windows software developers to get the best performance in their code.

Resources and References

Related White Papers:

Porting and Optimizing Multimedia Codecs for AMD64 Architecture on Microsoft Windows®, Winhec 2004 paper by Harsha Jagasia, Advanced Micro Devices.

Configuring Microsoft® Visual Studio Projects to support the AMD64 architecture, Winhec 2004 paper by Evandro Menezes, Advanced Micro Devices.

Application Software Considerations for NUMA-Based Systems, whitepaper from Microsoft on Microsoft.com or MSDN

Tools and resources from Microsoft:

- 64-bit OS versions: Server 2003 and Windows® XP Pro
- Platform SDK (PSDK) compiler, debugger, libraries, documentation
- Device Driver Developer Kit (DDK)
- Game SDK (DirectX)
- Windbg debugger
- The upcoming MS Visual Studio Whidbey release will feature full support for AMD64
- MSDN web site, and Microsoft.com
- Conferences: TechEd, Developer Days, WinHEC, PDC, ISV hands-on labs

Tools and resources from AMD:

- Developer Center: Sunnyvale CA, on-site and remote access to AMD64 systems; sign up via the AMD web site “Develop With AMD”
- CodeAnalyst tool: timer-based and event-based profiler, 32-bit and 64-bit code, easy to use, free download from AMD.com

Documentation and presentations:

- Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™
- AMD64 Architecture Programmers’ Manuals
- Numerous papers and presentations including
  Optimizing for the AMD Opteron Processor by Tim Wilkens Ph.D AMD
  Smashing the Barriers of 32-bit Gaming by Mike Wall, Sr. MTS AMD
  Porting Windows® Device Drivers to AMD64 Platforms AMD White Paper

  Look for “Develop with AMD” and “AMD64 Developer Resource Kit” on http://www.amd.com web site