Designing & Optimizing Software for N Cores

October, 2006

Michael Wall, Sr. MTS Software Engineer
Robin Maffeo, MTS Software Engineer
Justin Boggs, Sr. Developer Relations Engineer

Agenda

This talk is for PC Developers.
It's about maximizing performance.

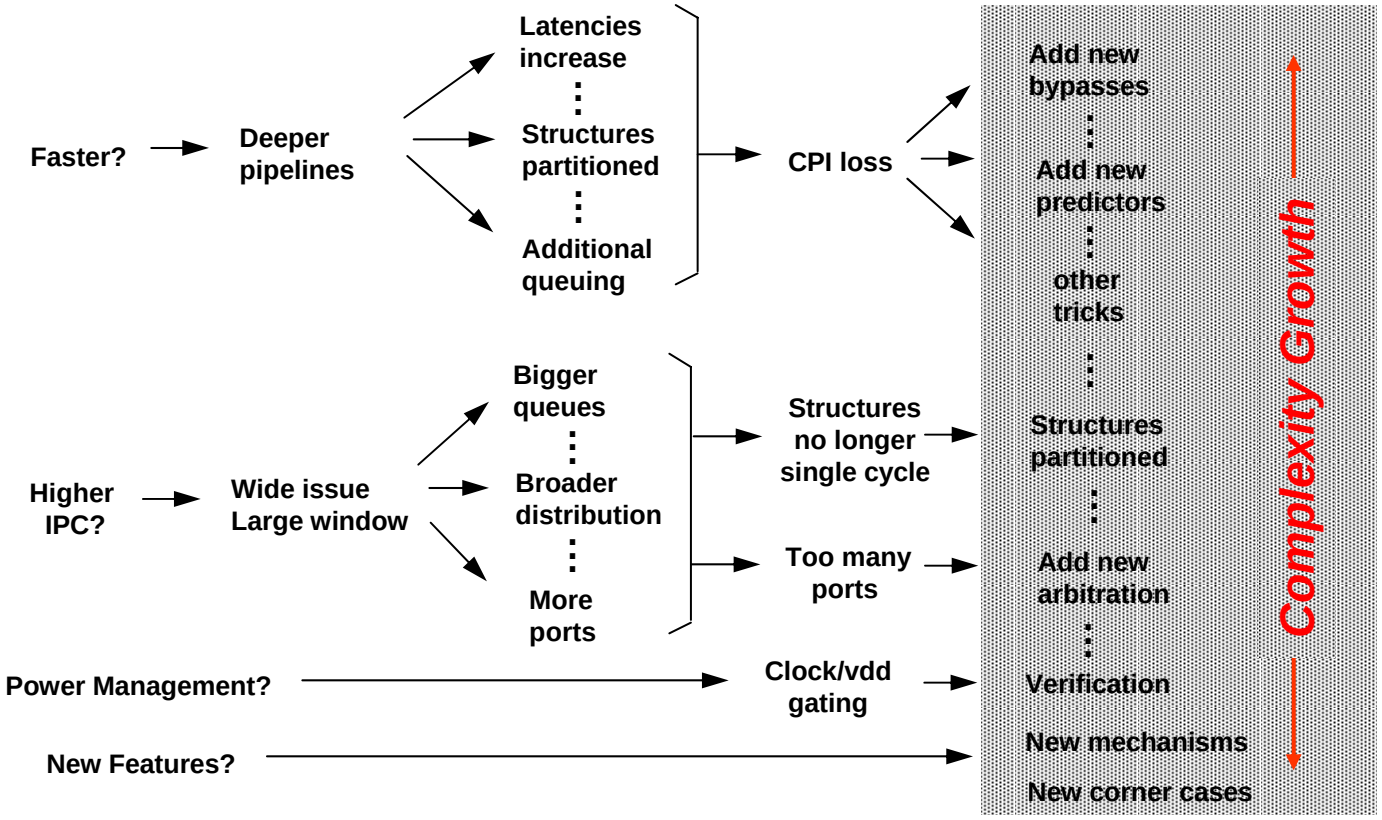
- Parallelism & Multi-threading (TLP)
- Multi-threading Design
- Threading Options on Windows®
- Threading Performance
- AMD Architecture & Threading
- Call To Action

Parallelism & Multi-threading (TLP)

A Closer Look at *Parallelism*

Instruction-level Parallelism (ILP)

- Executing multiple instructions from same program at the same time
- Superscalar hardware picks up most available ILP (*complexity effective*)



Superposition … *complexity tends to compound!*

A Closer Look at *Parallelism*

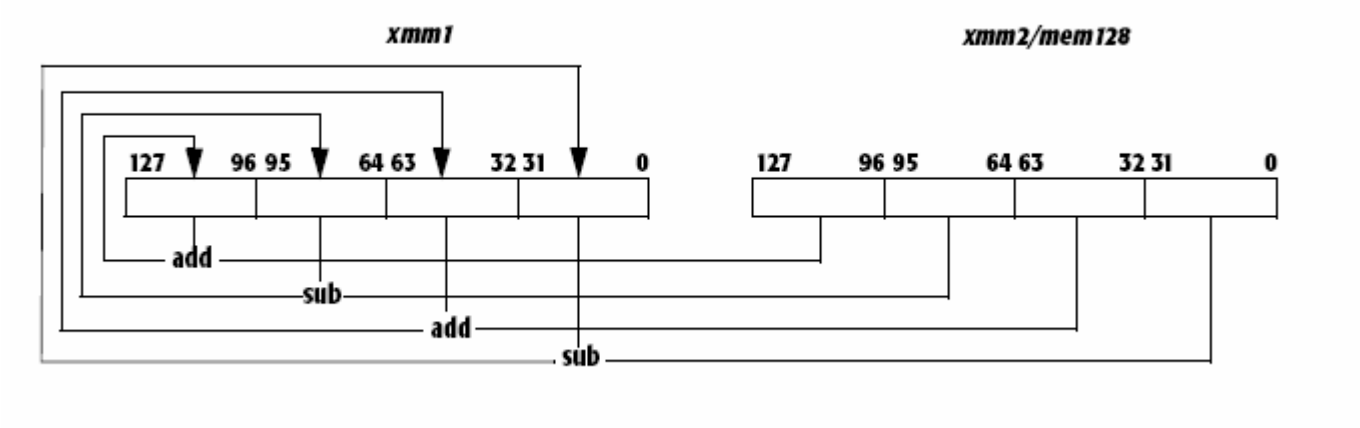
Instruction-level Parallelism (ILP)

- Executing multiple instructions from same program at the same time
- Superscalar hardware picks up most available ILP (*complexity effective*)

Data-level Parallelism (DLP)

- Executing same instruction on multiple pieces of data at the same time
- Vector-style processing -- SSE hardware operates in this manner

ADDSUBPS **Add and Subtract Packed Single-Precision**



A Closer Look at *Parallelism*

Instruction-level Parallelism (ILP)

- Executing multiple instructions from same program at the same time
- Superscalar hardware picks up most available ILP (*complexity effective*)

Data-level Parallelism (DLP)

- Executing same instruction on multiple pieces of data at the same time
- Vector-style processing -- SSE hardware operates in this manner

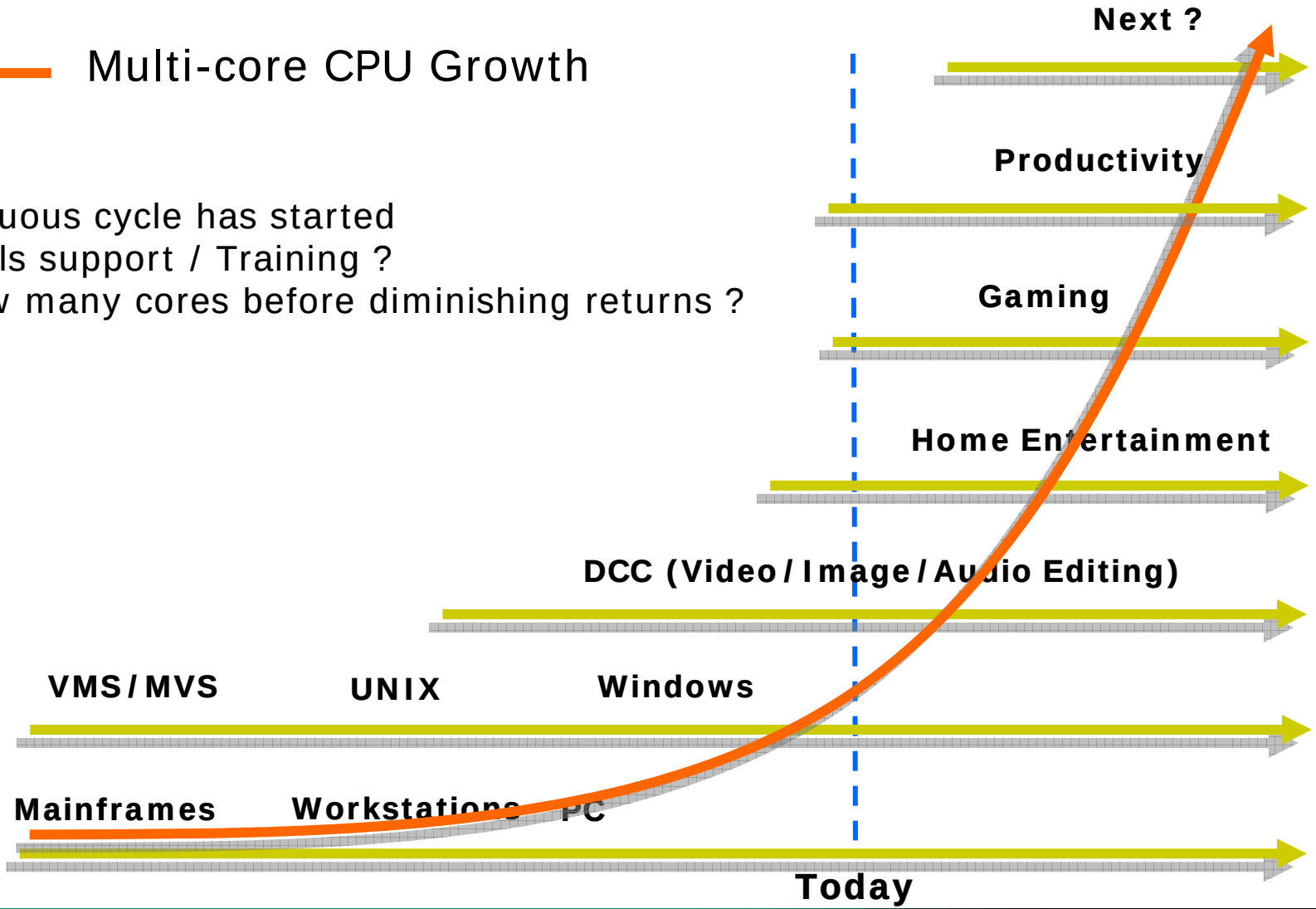
Thread-level Parallelism (TLP) – *several types:*

1. Concurrent Applications
   - Multiple programs running at the same time
   - Multiple OS's on virtualized hardware image
   - Collection of services integrated into a single "application"
2. Internet Transactional
   - Multiple computers running the same application
3. Parallel Applications - *The Holy Grail of computer science*
   - Single application partitioned to run as multiple threads

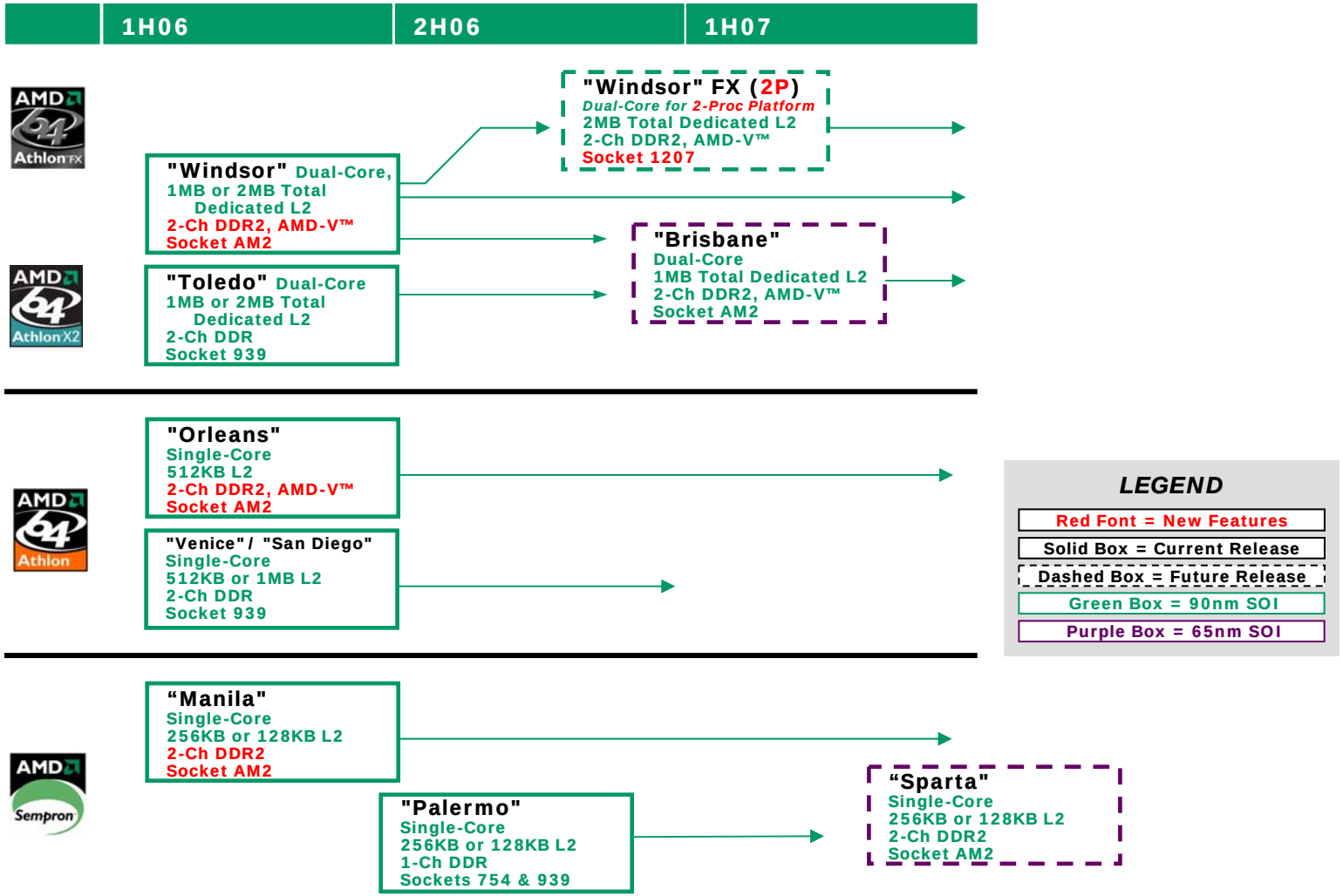
Multi-threaded Workloads

Multi-core CPU Growth

Virtuous cycle has started
Tools support / Training ?
How many cores before diminishing returns ?



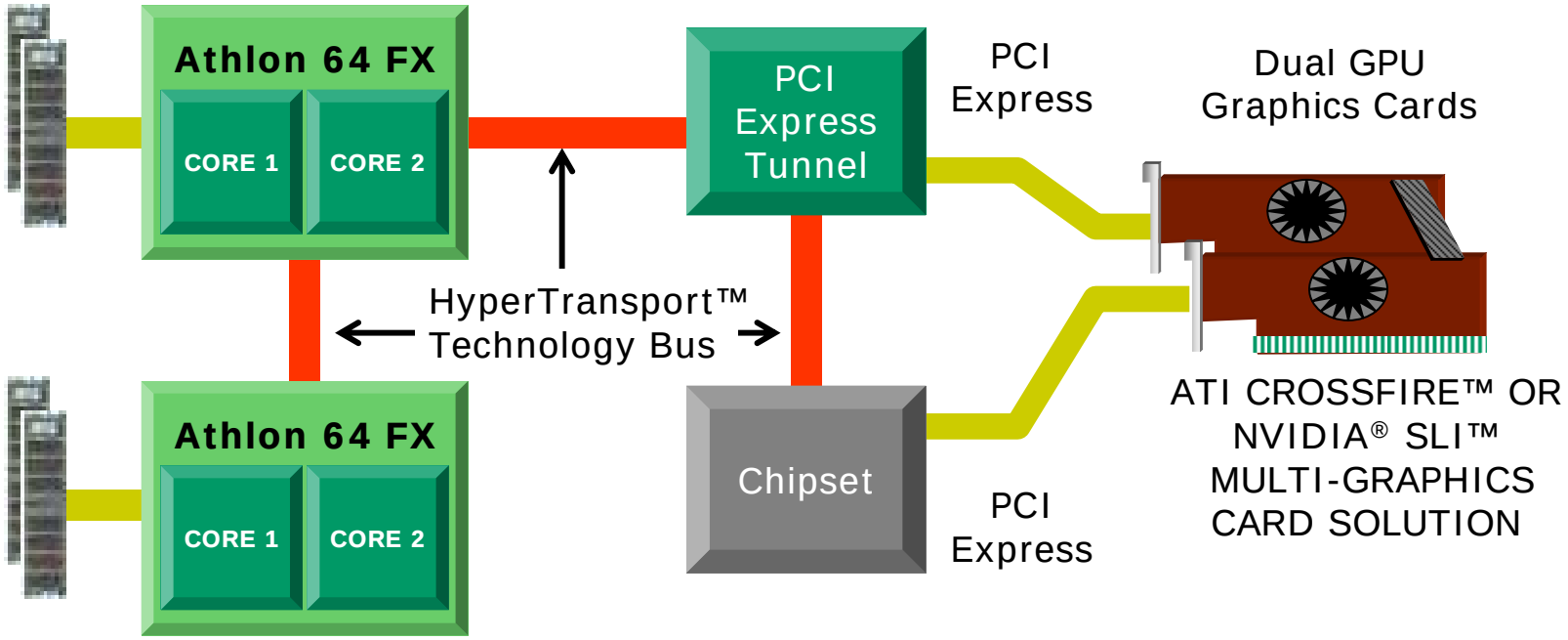
Desktop Processor Core Roadmap



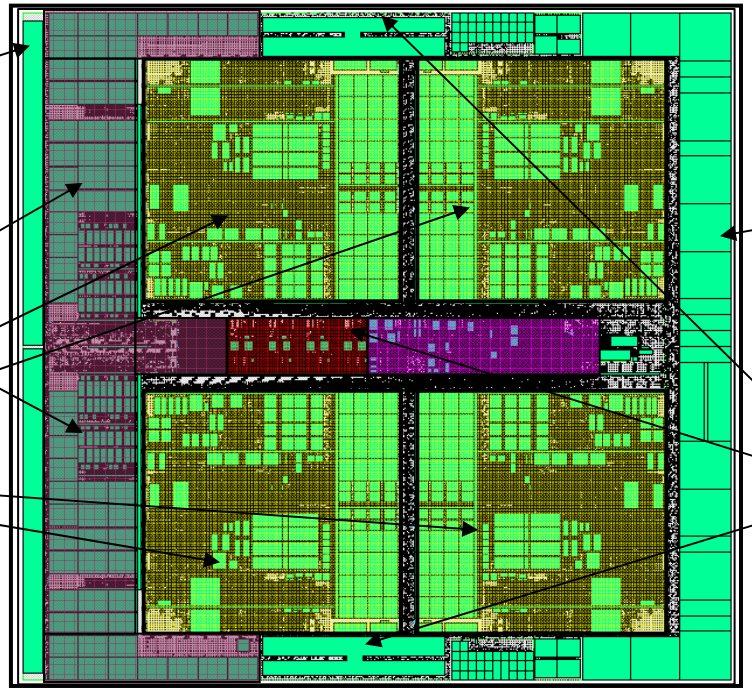
The “4x4” Platform

Fully leveraging the unique benefits of Direct Connect Architecture

Extending our legendary gaming and enthusiast platform performance



A Closer Look at AMD's Quad-Core CPU for '07



- 32B instruction fetch
- Enhanced branch prediction
- Out-of-order load execution
- Up to 4 DP FLOPS/cycle
- Dual 128-bit SSE dataflow
- Dual 128-bit loads per cycle
- Bit Manipulation extensions (LZCNT/POPCNT)
- SSE extensions (EXTRQ/INSERTQ, MOVNTSD/MOVNTSS)

- HT-1/3 links (Up to 5.2GT/sec)
- Enhanced crossbar
- DDR2 with migration path to DDR3
- FBDIMM when appropriate
- Enhanced power management
- Enhanced RAS

Multi-threading Design

What is Threading and Why is it Important?

- On Windows®, threads are the unit of work that is scheduled on a processor
- The “free lunch” is over with respect to significant increases in single-threaded execution
  - Single-core performance not increasing as in the past
  - Dual-core processors are proliferating everywhere
    - *Quad-core and larger are coming soon*
  - Without multi-threading, those cores will be underutilized, wasting otherwise usable CPU cycles

Multi-threading Design

- Ideally, threads should not be assigned per-function
  - Total number of threads is limited, doesn't scale
  - Cross thread communication and synchronization can be expensive and inefficient
  - Threads can stall each other
  - Poor load balancing
  - Producer/consumer threads have similar issues...
- The solution: Data-parallel threads
  - Multiple threads which do the same or similar work on multiple chunks of data
  - Number of threads not limited – can scale to the number of cores on the machine

Data-Parallel Threading

- Look for the most “heavyweight” task
  - Ensure it is suitable for out-of-order (i.e. parallel) execution
- Create threads (or use one from a pool) to partition data sets in the task across available cores
- Add sync points where necessary
  - Keep thread communication to a minimum
- Process all data and wait until all threads complete

- Repeat for other heavy tasks in the application

Threading Options on Windows®

I: Threading on Windows®: Win32 API

- Threads can be managed directly via the Win32 API. For example:
  - **`CreateThread()`**
  - **`ResumeThread()`**
  - **`GetThreadPriority()`** and **`SetThreadPriority()`**
  - **`SetThreadAffinityMask()`**
- You must do your own synchronization between threads with CriticalSections, Events…
  - Protect your data
  - Be aware of deadlocks and race conditions
  - Avoid Mutex – overkill for most situations, unless you are cross process
  - Too much synchronization can kill your performance, particularly as number of cores increases
- A thread in an executable that calls the C run-time library (CRT) should use the **`_beginthreadex`** and **`_endthreadex`** functions for thread management rather than **`CreateThread`** and **`ExitThread`**

I: Threading on Windows®: Win32 Example

```
hThread = CreateThread(
                NULL,           // Handle is not inherited
                0,              // Default Stack Size
                PhysicsThread, // Function to be executed
                pData,          // * to variable passed in
                0,              // Thread runs after creation
                NULL);          // Thread identifier not returned

DWORD WINAPI PhysicsThread(LPVOID lpParam)
{
   pPhysicsData = (PPHYSICSDATA) lpParam;
   while(TRUE)
   {
        WaitForSingleObject(startEvent, INFINITE);
        ... Party on physics data ...
   }
}
```

I: Threading on Windows®: Tuning

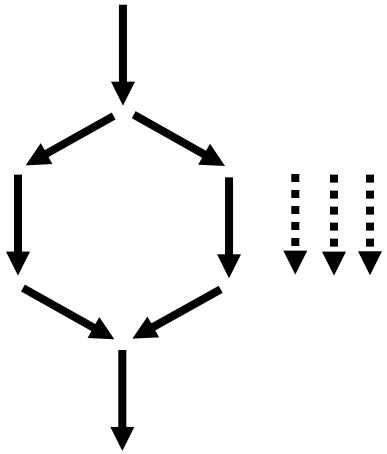
- Tune the number of threads you create based on number of processors in the system
- Generally, one thread per core is ideal
  - Except if your threads are not very busy and tend to block
  - Too many threads will cause excessive context switches, and hurt performance, so measure
- On Vista, XP x64, and Server 2003 use new **`GetLogicalProcessorInformation()`** API to obtain more detailed specifics on available processor cores and NUMA nodes (more on this later)

II: Threading on Windows®: Thread Pool

- Alternative to managing your own threads: the Windows Thread Pool
  - Good for many small work items that can be executed asynchronously
  - The burden of distributing work is still up to the developer
  - Windows will handle thread creation and destruction for you
  - Vista implementation was completely rewritten and has a new API: faster, supports multiple pools in a process, distinction between IO threads and non-IO threads removed

III: OpenMP Threading

- Simplest form of multi-threaded programming
  - One **#pragma** enables data parallel loop threading
  - Handles parallelization for you with “fork and join” programming model
  - If multiple cores are unavailable, serial execution occurs just as if OpenMP syntax wasn’t present



- Methodology and syntax reduce chance of complex threading bugs
  - Races, data corruption due to lack of protection…

- OpenMP support is new in Visual Studio® 2005
  - Underlying mechanism relies on Windows® Thread Pool to execute tasks

III: OpenMP: Example

```
#pragma omp parallel for
for (int i = 0; i < size; i++)
{
  x[i] = (y[i] + y[i+1]) / 2;
}
```

- Computes average of two values in one array, storing into another array
- Join at end of region is implicit – all threads complete
- OpenMP library handles scheduling over available cores in a static fashion (roughly iterations / cores)

III: OpenMP: Dynamic Scheduling

```
#pragma omp parallel for schedule(dynamic, 50)
for (int i = 0; i < size; i++)
{
  x[i] = (y[i] + y[i+1]) / 2;
}
```

- Dynamic scheduling helps adjust for imbalances
  - Good for varying or unpredictable work in loop body
- In this example, OpenMP assigns 50-iteration “chunks” across available threads until loop is complete
  - OpenMP internally uses synchronization to schedule threads

    Use large enough chunks to mitigate overhead – and then measure!

III: OpenMP: Rules of Thumb

- Thread your outer-most loop
  - Coarse-grained data parallelism
  - Minimum number of fork and join events
- Define local variables *inside* the loop
  - OpenMP will create a private copy for each thread
- Global variables
  - OK if the globals are *read only*
  - Writing to global variables will cause trouble
  - common but dangerous code:

```
*global++ = foo;   // danger! global pointer was
  modified
```

  - safe syntax:

```
*(global + local_offset++) = foo;
```

IV: Windows® Fibers

- Fibers allow the application to control scheduling of threads, allowing better scalability
  - Since the application knows when to switch tasks, rather than relying on the OS to switch threads, context switch overhead is lowered
- However, development cost can be high and is probably not worth it for the vast majority of applications
  - Requires you to effectively write your own scheduler

    Time to rewrite your application…
  - Windows thread scheduler keeps improving over time, mitigating fiber advantage

V: Lockless Programming

- Allows thread-safe operations without coarser locks (like critical section)

- "Lockless" programming can be very tricky
- It's very easy to have an implementation that looks correct, but breaks infrequently or only on certain processors
  - Bugs may be introduced that are subtle and very hard to find
  - This does not apply to simple scenarios – for example, **`InterlockedIncrement(refCount)`** is safe; complex dependent operations (using flags) are more problematic
- Avoid "clever" lockless mechanisms if you can – but sometimes it's not feasible to architect around a hot lock and scalability is paramount
  - If you must go lockless, know what you're doing
    Be aware of atomicity, processor memory models and load reordering behavior
  - In addition, the compiler can reorder instructions – use **`volatile`**, **`_ReadBarrier`**, and **`_WriteBarrier`** where necessary

AMD Suggestions

- Think about multi-threading and utilizing all cores at the beginning of your project
  - Bake it into your development, don't tack on later
- Separate or duplicate data to avoid synchronization
  - Consider the tradeoffs between working set and code simplicity/scalability
  - Use double buffering (read state from one buffer, write to the other, then swap for next frame)

    Allows state to be broken across multiple threads
  - Where practical use OpenMP to reduce complexity and distribute work across cores

    Explicit synchronization not needed in many situations

Threading Links

- For more on OpenMP:
  - http://www.openmp.org
  - http://msdn.microsoft.com/msdnmag/issues/05/10/OpenMP
- Concurrent programming:
  - http://msdn.microsoft.com/msdnmag/issues/05/10/ConcurrentAffairs
  - http://msdn.microsoft.com/msdnmag/issues/05/08/Concurrency

Threading Performance

Overhead of Windows® Threading Options

- Measurements on AMD platforms
  - CreateThread:
    - Takes ~160k cycles
  - OpenMP parallel loop:
    - Adds about 20k CPU cycles for a fork-and-join on dual-core
    - Always balance workloads across threads!
  - CreateThread and WaitForMultipleObjects for loop:
    - Same approximate overhead as OpenMP, with balanced workloads

Overhead of Windows® Threading Options

- Measurements on AMD platforms
  - Lockless - InterlockedXXX operation:
    - Atomic read/modify/write operation implemented in hardware
    - MemoryBarrier macro in VS2005 prevents compiler from re-ordering
    - Declare thread sync flags as **volatile** to ensure they're written to memory
    - Very little overhead for the actual Interlockedxxx operation
      - But another thread my be spin-locking while waiting for it
      - Minimize data dependencies
  - Mutex:
    - Slow, avoid it
  - Critical Section:
    - Faster than a Mutex, but still try and design to minimize these

AMD Architecture & Threading

AMD Architecture Considerations for Multi-threading

- Cache
- NUMA
- Detecting the AMD processor cores
- XP vs. Vista differences

Cache Can Largely Impact Performance

- Maximize benefit from the cache in dual-core and beyond
- Each AMD Athlon™ 64 processor or AMD Opteron™ Processor core has its own separate L1 caches
- ***Each core also has its own separate L2 cache***
- Greater parallelism enables added performance:
  - L2 cache latency does not increase when another L2 is added
  - Total cache bandwidth scales up linearly with the number of cores
  - Threads cannot evict another thread's data from other L2
- L2-per-core gives *symmetric* scaling beyond dual-core

Cache Impact on Threading Architecture

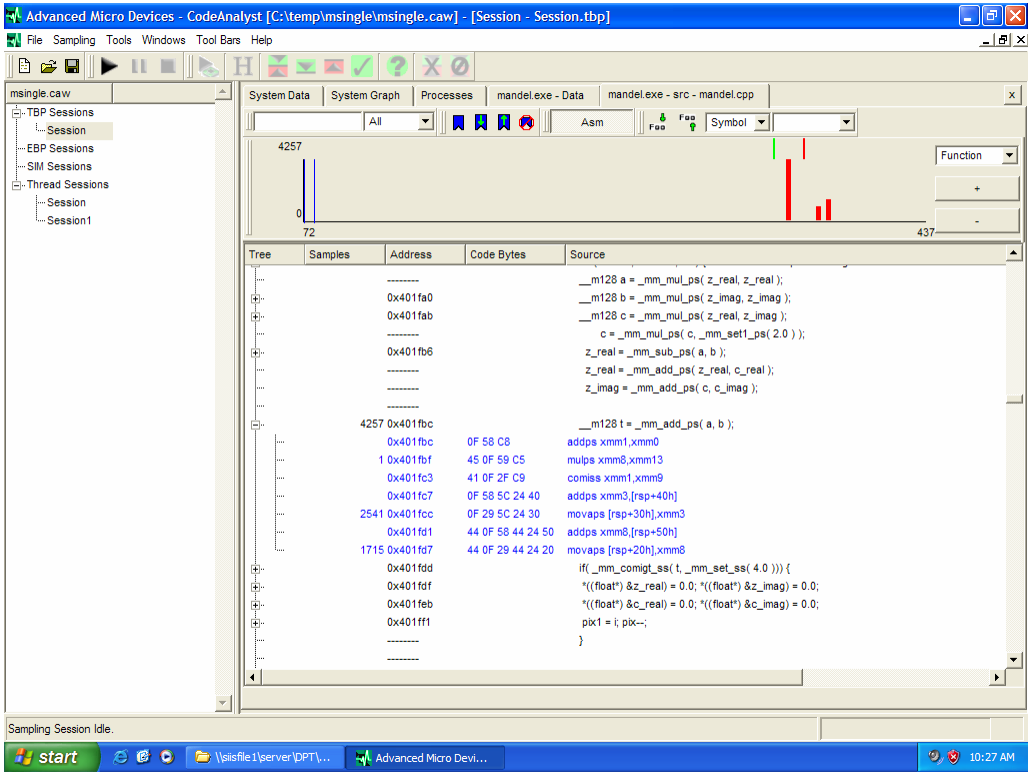
- Avoid “cache thrashing” between cores
  - Avoid multiple threads writing to the same variables (of course)
  - Also avoid one thread frequently writing what another is reading
    - This is just a standard rule of threaded programming
  - Beware of **false sharing** which can cause thrashing:
    - Two threads modifying **different variables** which occupy the **same cache line**
      - Can happen in heap data or in global variables
    - One safe approach: **`_aligned_malloc`** for 64-byte aligned heap chunks
    - Another handy alignment trick: **`__declspec(align(64))`**
    - See MSDN for complete details on managing alignment

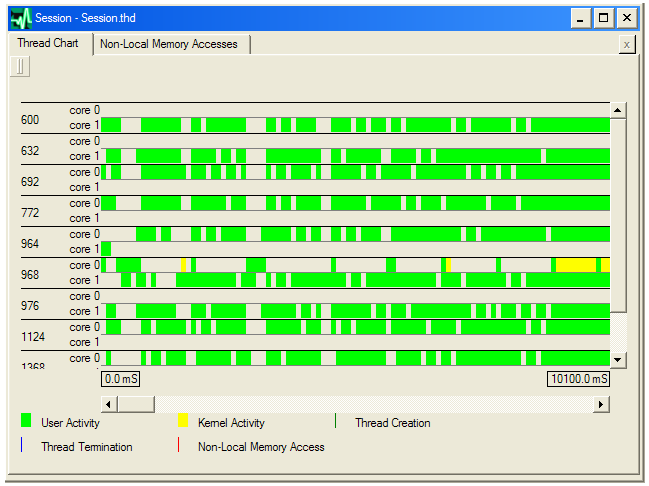
- Use AMD CodeAnalyst™ profiler to examine threads and cache events
  - Thrashing would appear as excessive cache refill events
  - Just one way AMD CodeAnalyst can help you build faster code
  - Available at http://developer.amd.com/

AMD CodeAnalyst™ Profiler

Use AMD CodeAnalyst, profile your 32 and 64-bit code!

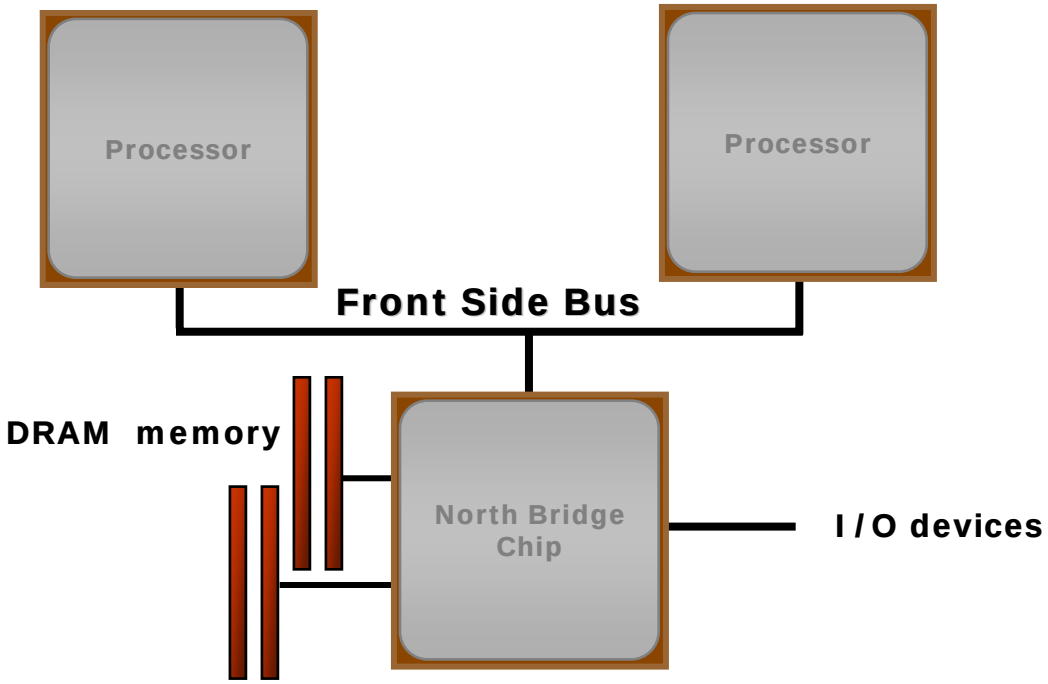
- Timer-based & event-based profiler
- Integrates with the VS2005 IDE





- *Thread* Profiler shows a thread execution timeline

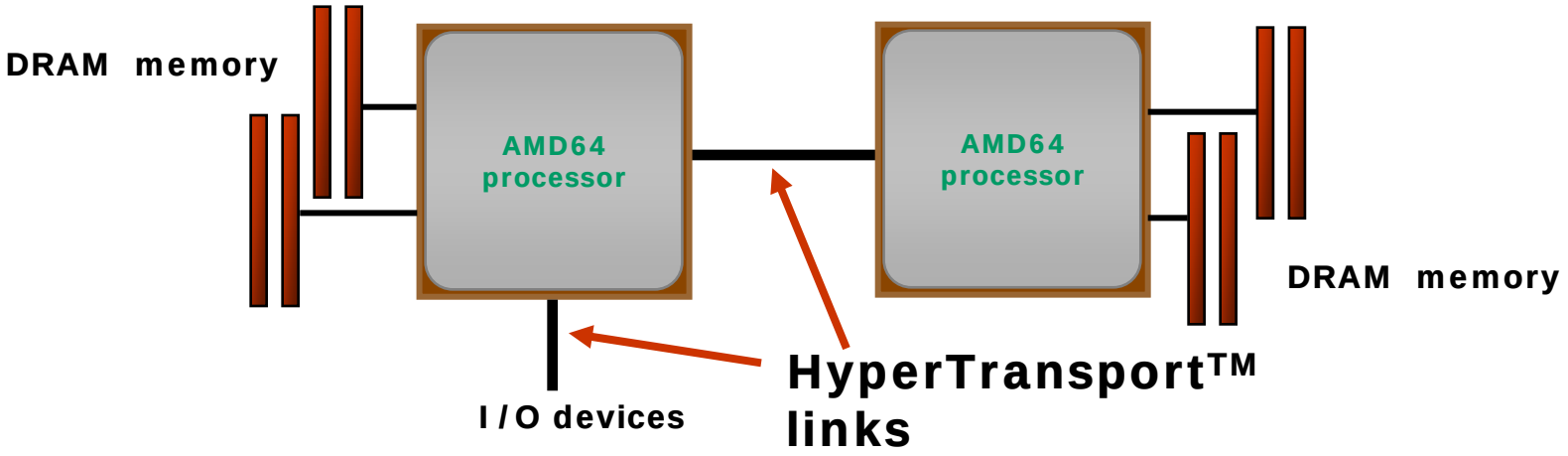
Multiprocessing: “The Old Way”



- The old Front Side Bus = Front Side Bottleneck!
- CPUs must wait for each other, memory and I/O
- North Bridge chip slows memory access

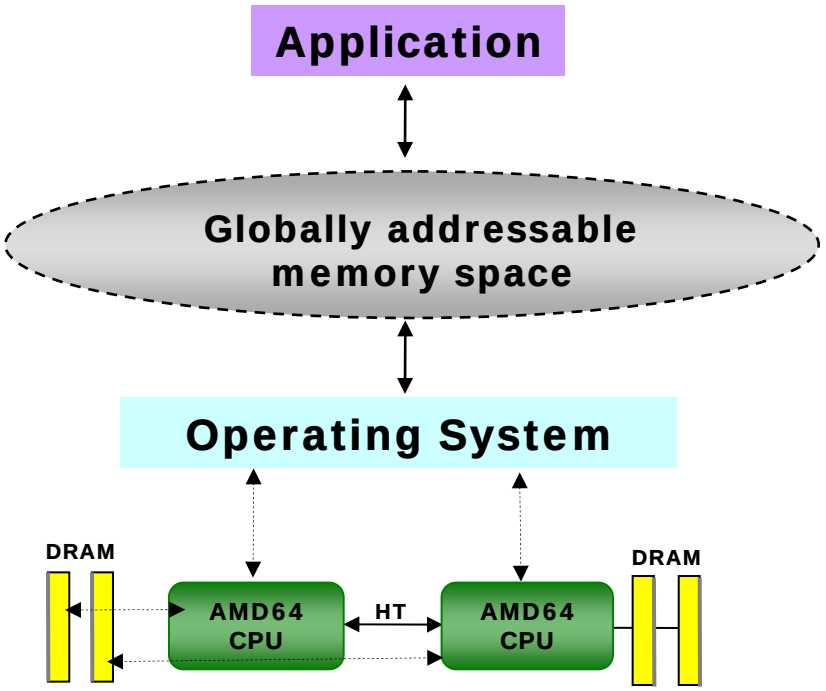
AMD64 Technology

Glueless MP, AMD's Direct Connect Architecture



- HyperTransport™ technology links connect CPUs, and I/O
- No extra silicon required for multiprocessing!
- Super-low latency to local memory banks
- Now I/O has its own separate link(s)
- Memory bandwidth, and capacity, scale up with added CPUs

Non-Uniform Memory Access (NUMA)



Memory initialized into globally addressable memory space with processors maintaining cache coherency across this space

OS assigns threads from same process to the same NUMA node

Processor has local memory and uses HyperTransport™ technology for high-speed access to non-local memory

Windows® ccNUMA support

Microsoft® Windows OS fully supports the AMD Opteron™ NUMA system architecture

- ccNUMA = cache coherent NUMA
- Relevant in multi-socket systems with more than one memory controller

- Windows XP Pro x64 and Server 2003 SP1 support NUMA
- Windows Vista and beyond will have the best NUMA support

- Memory affinity is optimized
  - The OS allocates physically local memory when possible
- ccNUMA APIs can be used for additional control
  - GetNumaNodeProcessorMask, SetThreadAffinityMask, etc.

ccNUMA: thread affinity on Windows®

- Typically just let Windows manage affinity
  - The OS knows everything
- Sometimes explicit affinity calls will improve performance
  - Be very careful
  - Don't assume anything about the NUMA topology
  - Don't make the affinity more strict than necessary
- NUMA APIs are documented at MSDN
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/numa_support.asp

ccNUMA: allocating memory on Windows®

- Windows XP Pro x64 and Server 2003
  - Create thread in suspended state
  - Set thread affinity to a particular NUMA node
  - Start thread
  - Allocate your memory
  - Initialize the memory to cause page faults (first touch)

    Then the OS physically commits the memory

- Windows Vista/Longhorn is slightly different
  - Create thread in suspended state
  - Set thread affinity to a particular NUMA node
  - Start thread
  - Allocate your memory
  - No need to initialize the memory (but no harm done)

    The OS physically commits the memory on allocate

Detecting the # of Cores Can Be Tricky

- Anticipate the adoption of virtual environments
  - Physical computer may be partitioned: you don't get the whole thing
  - Trust the OS to tell you about available resources
    - *Really try to avoid using low-level instructions on the hardware*
- For a recent OS (Vista, Windows® XP Pro x64, Windows Server 2003) use the new function **`GetLogicalProcessorInformation()`**
  - Gives you more complete info about NUMA, cache, etc.
  - Lets you distinguish true cores from SMT threads (pseudo-core)
- For Windows XP / 2000 **`GetSystemInfo()`** tells you how many processors you have

Performance Analysis Pitfalls

Do not use the “RDTSC” timestamp counter!

- “RDTSC” timestamp instruction works just the same as always…
  - … but each core has its own timestamp counter in a multi-core system…
  - … and they are *not* guaranteed to be in sync…
  - … therefore **comparing two values from RDTSC can be dangerous.**
- Windows will move your process between cores
- Use:
  `QueryPerformanceCounter()`
  `QueryPerformanceFrequency()`
- See “Game Timing and Multicore Processors” at MSDN:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Game_Timing_and_Multicore_Processors.asp

Windows® XP vs. Vista

- Vista has:
  - Better NUMA support
  - Better thread and process scheduler
  - Improved heap manager in an SMP environment
  - Superior resource detection: **`GetLogicalProcessorInformation()`**
  - A new thread pool API and implementation

Call To Action

Call To Action

- **Think about Multi-Threading at the beginning of your project**
- **Use Data Parallel Threading as your threading model for scalable performance for years to come**
- **Develop, Test, and Optimize on AMD processor-based systems… Run Fast Everywhere!**

Trademark Attribution

AMD, the AMD Arrow logo, and combinations thereof, AMD Athlon, AMD Opteron, and AMD-V are trademarks of Advanced Micro Devices, Inc.
Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions.
HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.
Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.