
GDC
09
learn
network
inspire
www.GDConf.com
Game Developers Conference®
March 23-27, 2009 | Moscone Center, San Francisco

Lockless Programming in Games

Bruce Dawson
Principal Software Design Engineer
Microsoft
Windows Client Performance

Agenda

- Locks and their problems
- Lockless programming – a different set of problems!
- Portable lockless programming
- Lockless algorithms that work
- Conclusions

- Focus is on improving intuition on the *reordering* aspects of lockless programming

Cell phones

- Please turn off all cell phones, pagers, alarm clocks, crying babies, internal combustion engines, leaf blowers, etc.

Mandatory Multi-core Mention

- Xbox 360: six hardware threads
- PS3: nine hardware threads
- Windows: quad-core PCs for $500

- Multi-threading is mandatory if you want to harness the available power
- Luckily it's easy
  - As long as there is no sharing of non-constant data
- Sharing data is tricky
  - Easiest and safest way is to use OS features such as locks and semaphores

Simple Job Queue

- Assigning work:

```
EnterCriticalSection( &workItemsLock );
workItems.push( workItem );
LeaveCriticalSection( &workItemsLock );
```

- Worker threads:

```
EnterCriticalSection( &workItemsLock );
WorkItem workItem = workItems.front();
workItems.pop();
LeaveCriticalSection( &workItemsLock );
DoWork( workItem );
```

The Problem With Locks…

- Overhead – acquiring and releasing locks takes time
  - So don't acquire locks too often
- Deadlocks – lock acquisition order must be consistent to avoid these
  - So don't have very many locks, or only acquire one at a time
- Contention – sometimes somebody else has the lock
  - So never hold locks for too long – contradicts point 1
  - So have lots of little locks – contradicts point 2
- Priority inversions – if a thread is swapped out while holding a lock, progress may stall
  - Changing thread priorities can lead to this
  - Xbox 360 system threads can briefly cause this

THINK
GDC 09
learn network inspire
www.GDConf.com

Sensible Reaction

- Use locks carefully
  - Don't lock too frequently
  - Don't lock for too long
  - Don't use too many locks
  - Don't have one central lock
- Or, try lockless

Lockless Programming

- Techniques for safe multi-threaded data sharing without locks
- Pros:
  - May have lower overhead
  - Avoids deadlocks
  - May reduce contention
  - Avoids priority inversions
- Cons
  - Very limited abilities
  - Extremely tricky to get right
  - Generally non-portable

Job Queue Again

- Assigning work:

```
EnterCriticalSection( &workItemsLock );
workItems.push( workItem );
LeaveCriticalSection( &workItemsLock );
```

- Worker threads:

```
EnterCriticalSection( &workItemsLock );
WorkItem workItem = workItems.front();
workItems.pop();
LeaveCriticalSection( &workItemsLock );
DoWork( workItem );
```

Lockless Job Queue #1

- Assigning work:

```
EnterCriticalSection( &workItemsLock );
InterlockedPushEntrySList( workItem );
LeaveCriticalSection( &workItemsLock );
```

- Worker threads:

```
EnterCriticalSection( &workItemsLock );
WorkItem workItem =
        InterlockedPopEntrySList();
LeaveCriticalSection( &workItemsLock );
DoWork( workItem );
```

Lockless Job Stack #1

- Assigning work:

```
InterlockedPushEntrySList( workItem );
```

BROKEN on Xbox 360!!!

- Wor

```
WorkItem workItem =
        InterlockedPopEntrySList();
```

```
DoWork( workItem );
```

Lockless Job Queue #2

- Assigning work – one writer only:

```
if( RoomAvail( readPt, writePt ) ) {
    CircWorkList[ writePt ] = workItem;
                                      ;
```

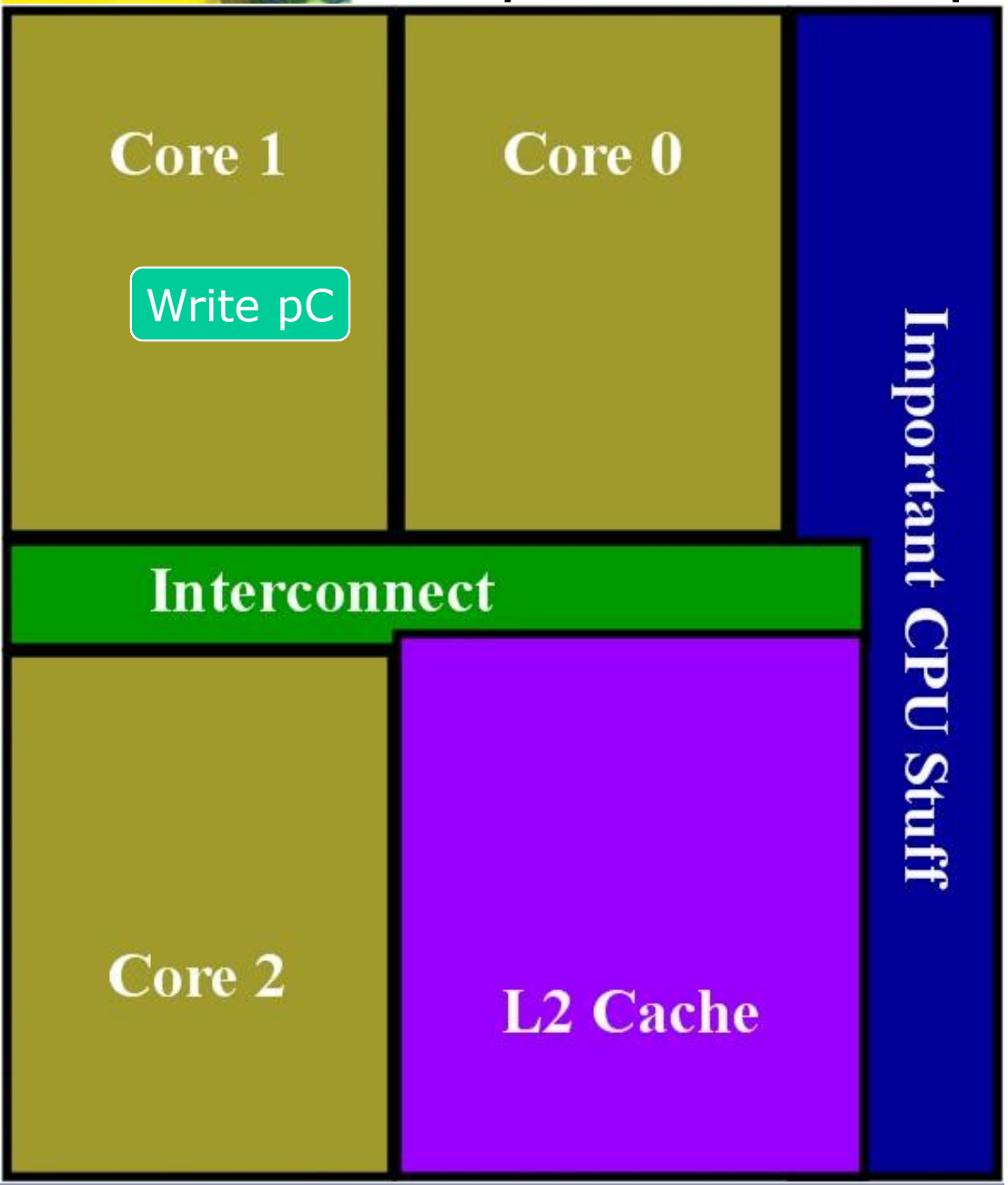
- Wor

```
if(                                    {
                  CircWorkList[ readPt ];
    readPt = WRAP( readPt + 1 );
DoWork( workItem );
```

Broken As Executed

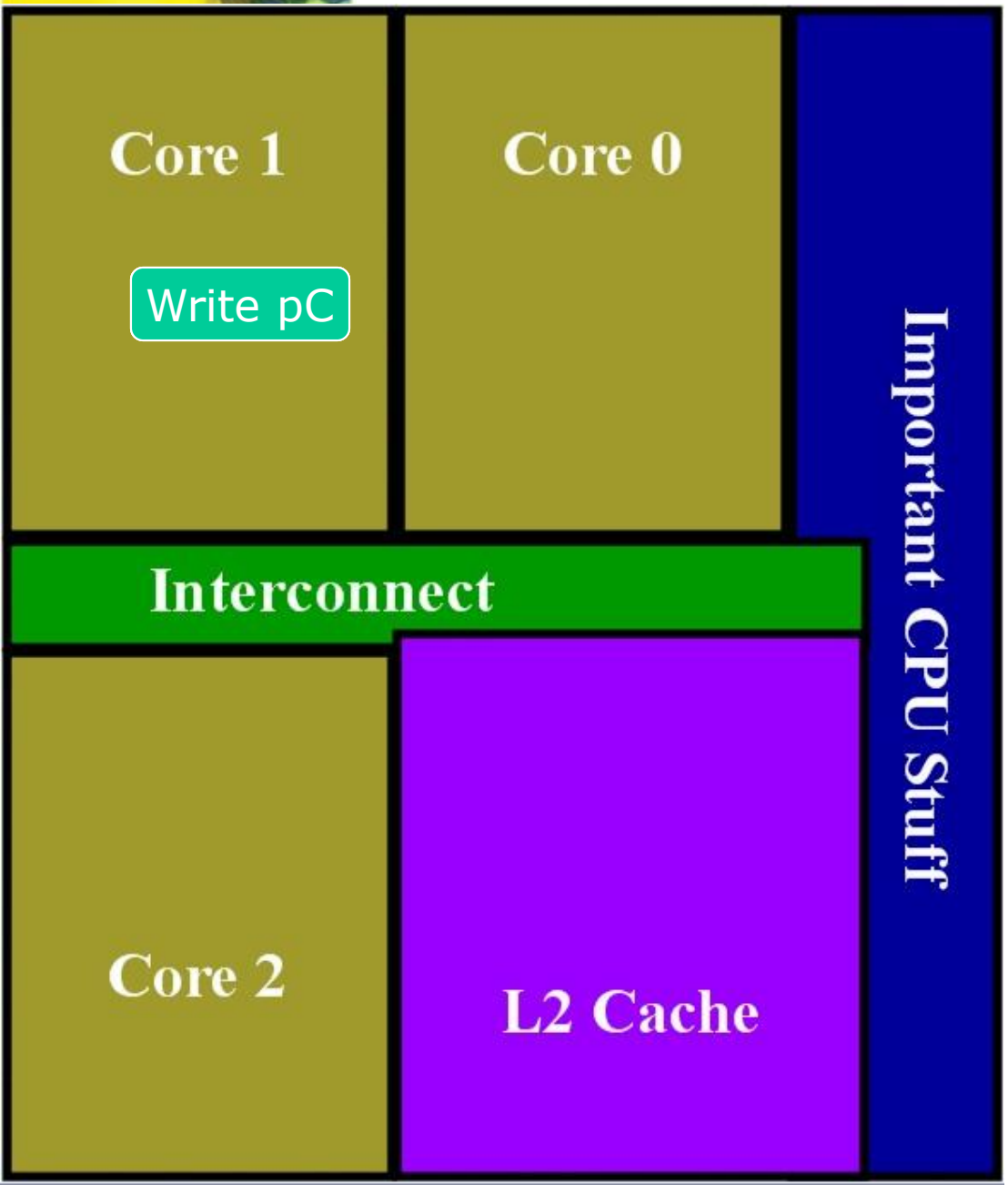
THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

Simple CPU/Compiler Model



Read pC
Write pA
Write pB
Read pD
Write pC

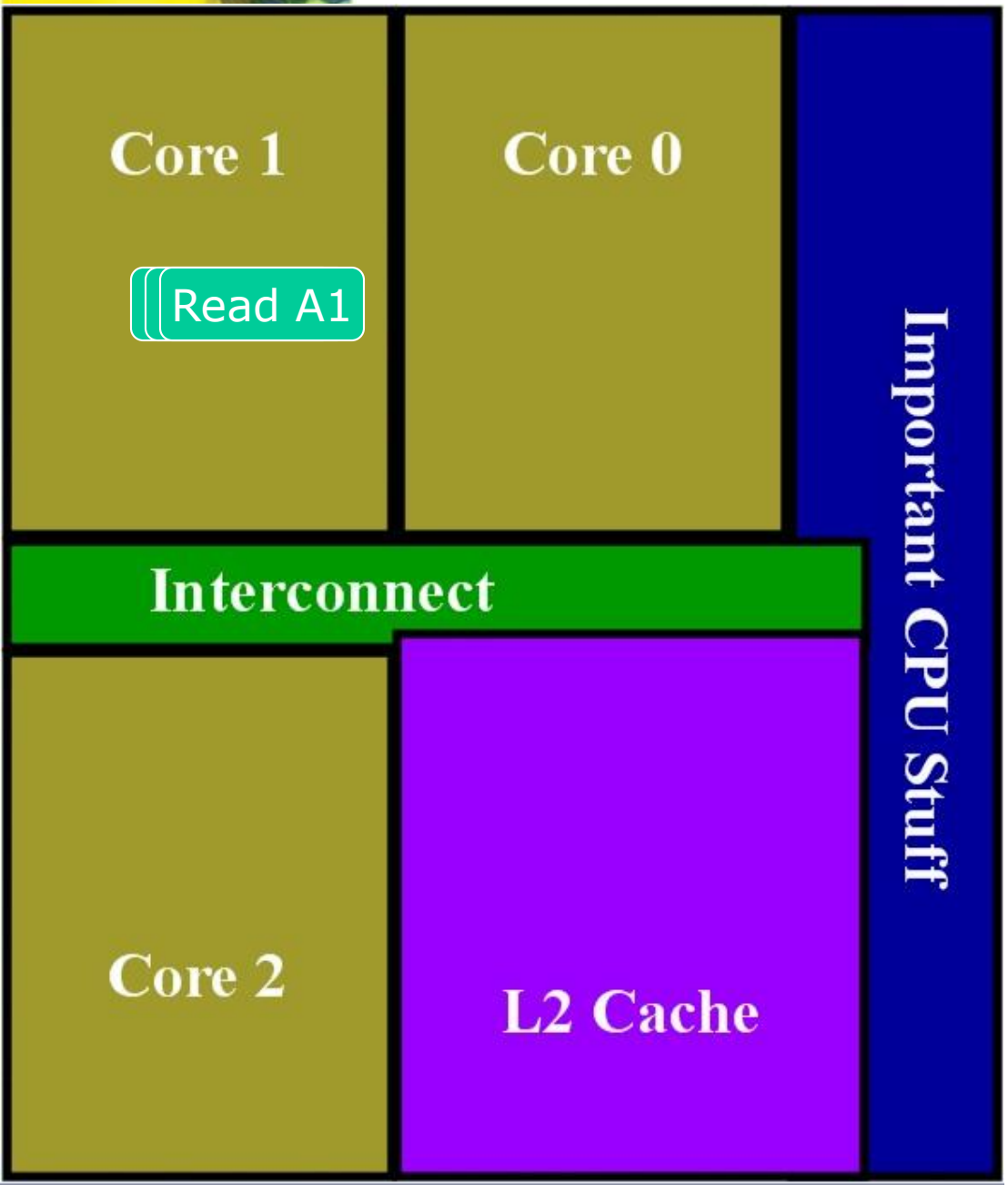
Alternate CPU Model



Write pA
Write pB
Write pC

Visible order:
Write pA
Write pC
Write pB

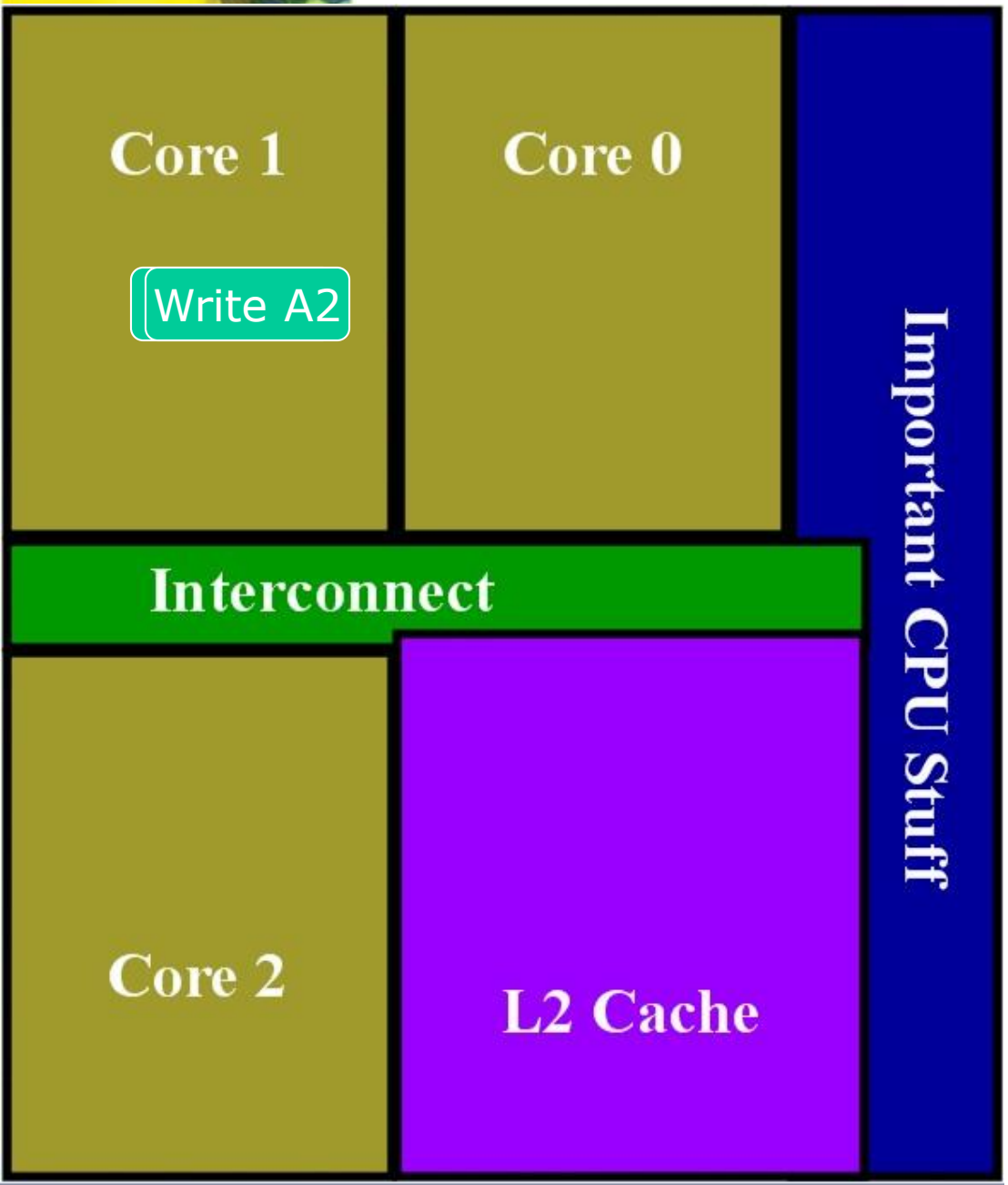
Alternate CPU – Reads Pass Reads



Read A1
Read A2
Read A1

Visible order:
Read A1
Read A1
Read A2

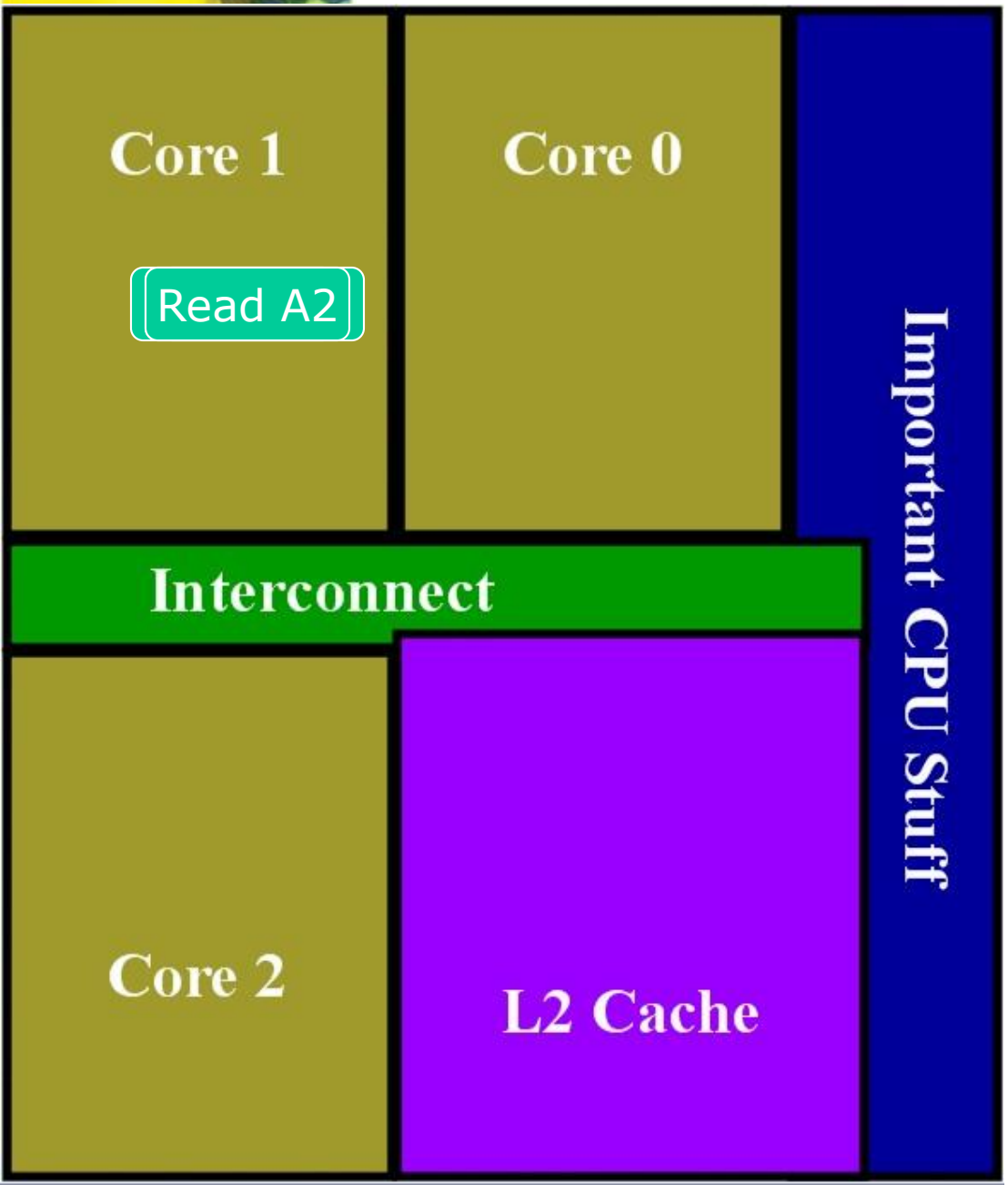
Alternate CPU – Writes Pass Reads



Read A1
Write A2

Visible order:
Write A2
Read A1

Alternate CPU – Reads Pass Writes



Read A1
Write A2
Read A2
Read A1

Visible order:
Read A1
Read A1
Write A2
Read A2

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

Memory Models

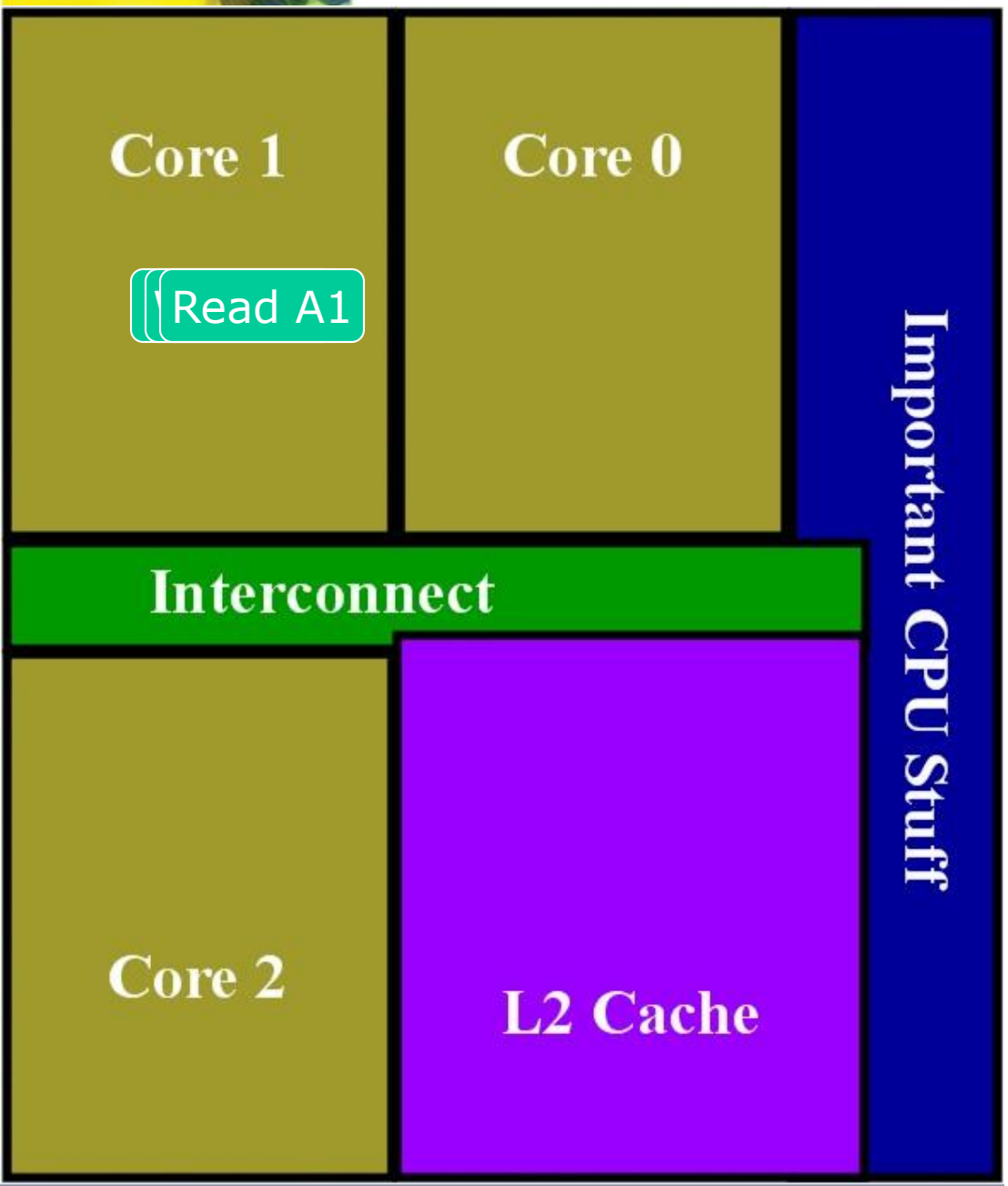
	x86/x64	PowerPC	ARM	IA64
store can pass store?	No	Yes*	Yes*	Yes*
load can pass load?	No	Yes	Yes	Yes
store can pass load?	No	Yes	Yes	Yes
load can pass store?**	Yes	Yes	Yes	Yes

- "Pass" means "visible before"
- Memory models are actually more complex than this
  - May vary for cacheable/non-cacheable, etc.
- This *only* affects multi-threaded lock-free code!!!

* Only stores to different addresses can pass each other

** Loads to a previously stored address will load that value

Improbable CPU – Reads *Don't* Pass Writes



Read A1
Write A2
Read A1

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

Reads Must Pass Writes!

- Reads not passing writes would mean L1 cache is frequently disabled
  - Every read that follows a write would stall for shared storage latency
- Huge performance impact
- Therefore, on x86 and x64 (on *all* modern CPUs) reads can pass writes

Reordering Implications

- Publisher/Subscriber model
- Thread A:

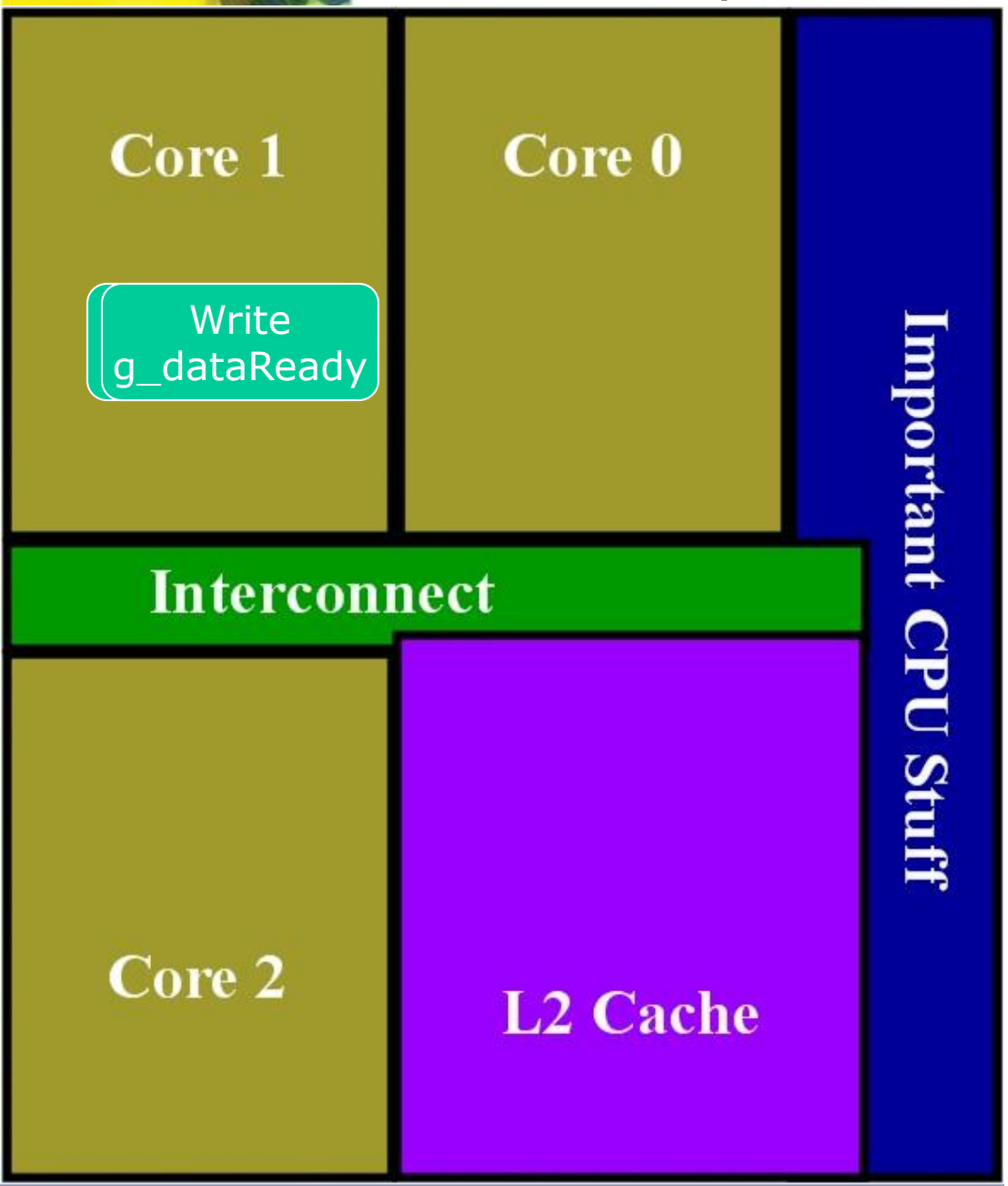
```
g_data = data;
g_dataReady = true;
```

- Thread B:

```
if( g_dataReady )
    process( g_data );
```

- Is it safe?

Publisher/Subscriber on PowerPC



Proc 1:

Write g_data

Write g_dataReady

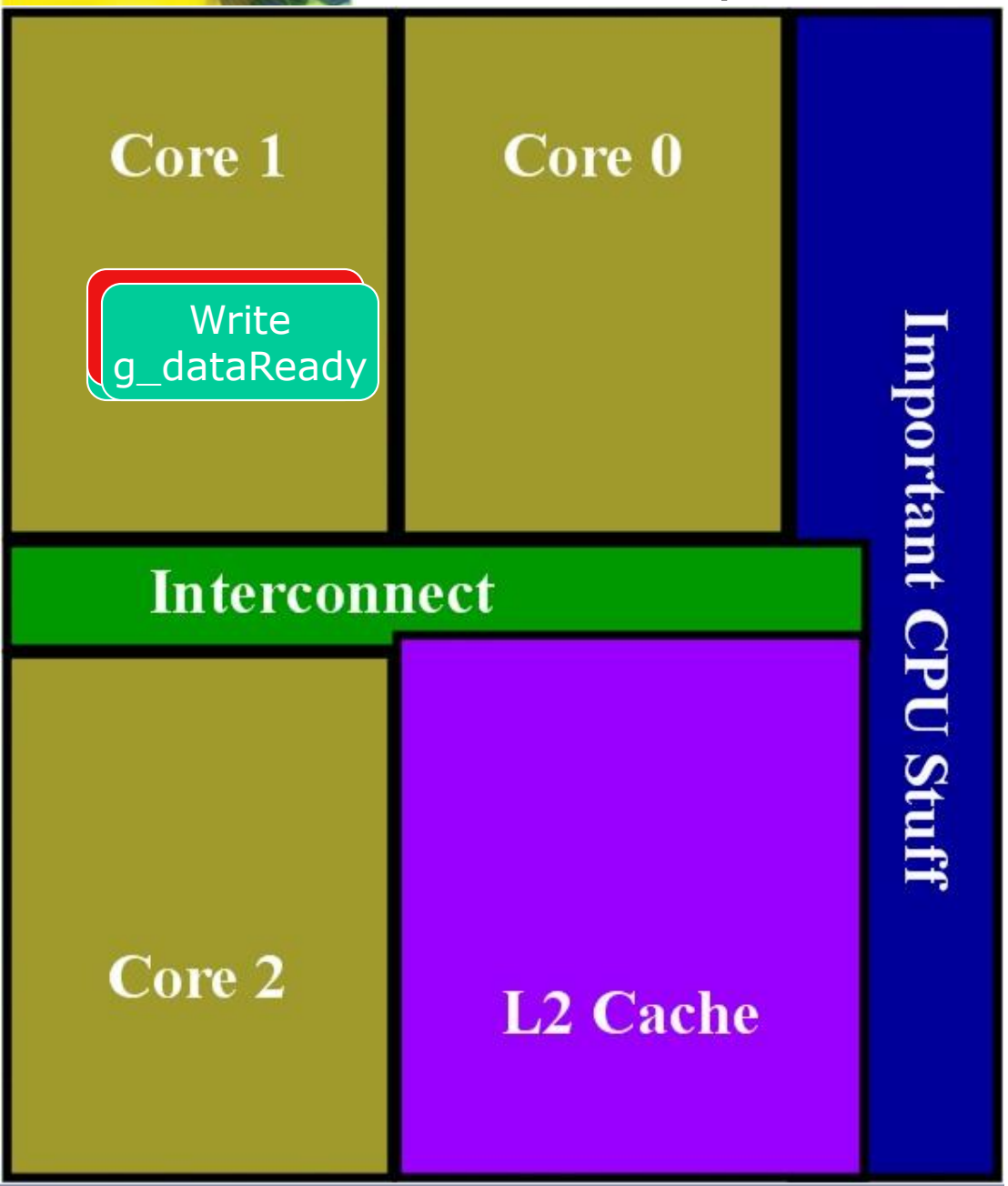
Proc 2:

Read g_dataReady

Read g_data

- Writes may reach L2 out of order

Publisher/Subscriber on PowerPC



Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

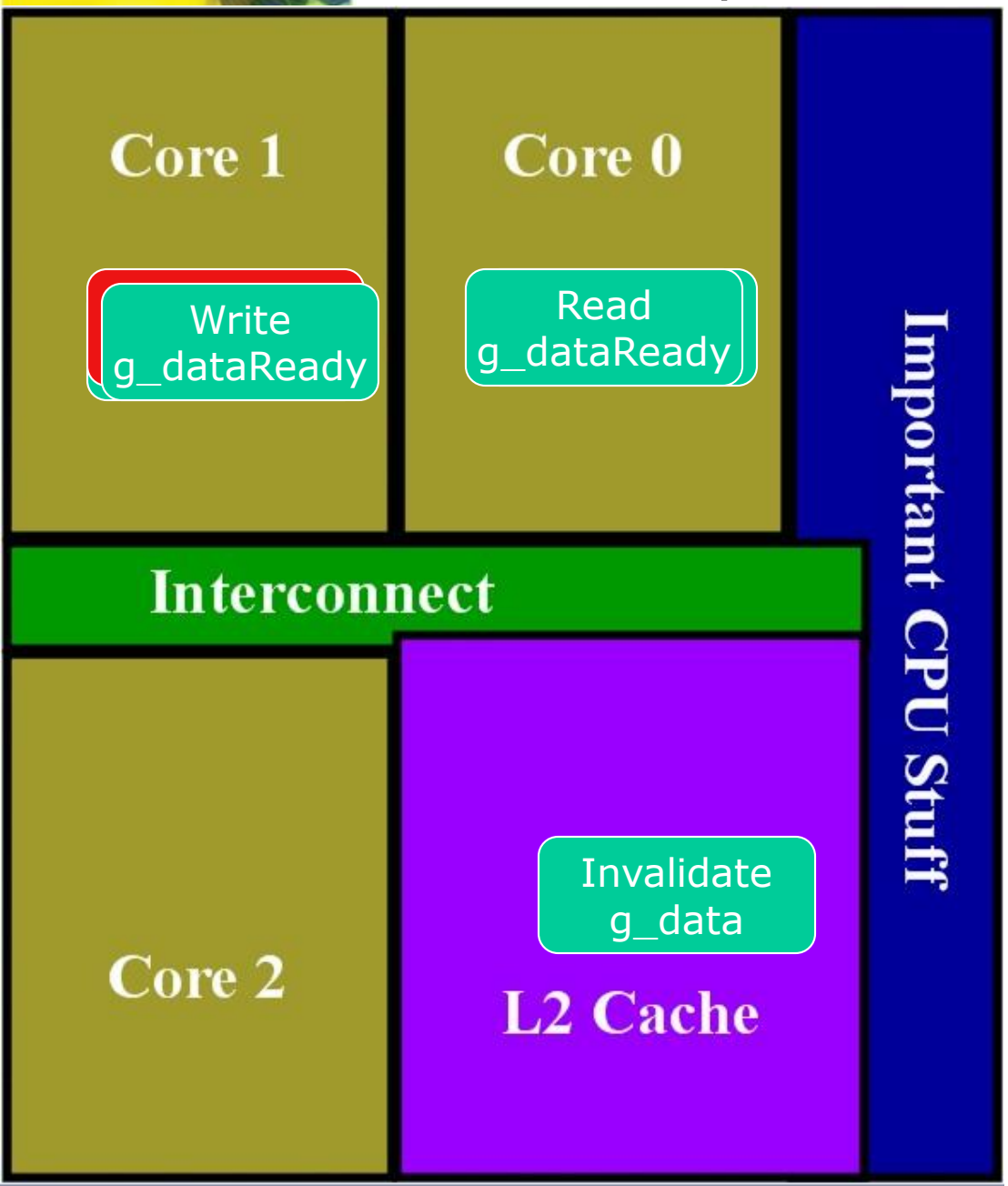
Proc 2:

Read g_dataReady

Read g_data

- Writes now reach L2 in order

Publisher/Subscriber on PowerPC



Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

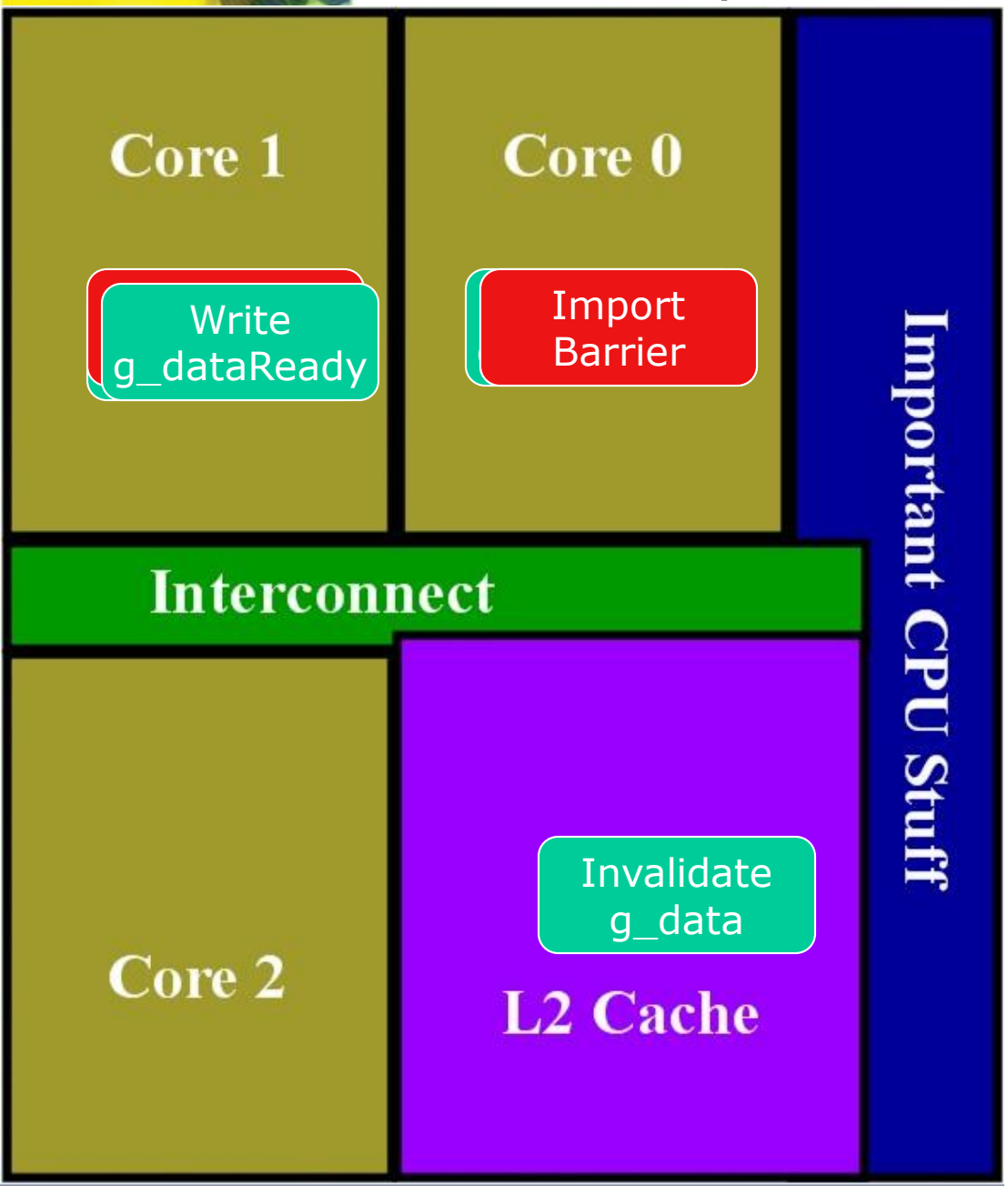
Proc 2:

Read g_dataReady

Read g_data

- Reads may leave L2 out of order – g_data may be stale

Publisher/Subscriber on PowerPC



Proc 1:

Write g_data

MyExportBarrier();

Write g_dataReady

Proc 2:

Read g_dataReady

MyImportBarrier();

Read g_data

- It's all good!

x86/x64 FTW!!!

- Not so fast…

- Compilers are just as evil as processors
- Compilers will rearrange your code as much as legally possible
  - And compilers assume your code is single threaded
- Compiler and CPU reordering barriers needed

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

MyExportBarrier

- Prevents reordering of writes by compiler *or* CPU
  - Used when handing out access to data
- x86/x64: _ReadWriteBarrier();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync();
  - Hardware barrier, prevents CPU write reordering
- ARM: __dmb(); // Full hardware barrier
- IA64: __mf(); // Full hardware barrier

- Positioning is crucial!
  - Write the data, MyExportBarrier, write the control value
- Export-barrier followed by write is known as write-release semantics

MyImportBarrier();

- Prevents reordering of reads by compiler *or* CPU
  - Used when gaining access to data
- x86/x64: _ReadWriteBarrier();
  - Compiler intrinsic, prevents compiler reordering
- PowerPC: __lwsync(); or isync();
  - Hardware barrier, prevents CPU read reordering
- ARM: __dmb(); // Full hardware barrier
- IA64: __mf(); // Full hardware barrier

- Positioning is crucial!
  - Read the control value, MyImportBarrier, read the data
- Read followed by import-barrier is known as read-acquire semantics

Fixed Job Queue #2

- Assigning work – one writer only:

```
if( RoomAvail( readPt, writePt ) ) {
    MyImportBarrier();
    CircWorkList[ writePt ] = workItem;
    MyExportBarrier();
    writePt = WRAP( writePtr + 1 );
```

- Worker thr

```
if( DataA
    MyImp
    WorkItem workItem =
                CircWorkList[ readPt ];
    MyExportBarrier();
    readPt = WRAP( readPt + 1 );
DoWork( workItem );
```

Correct!!!

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

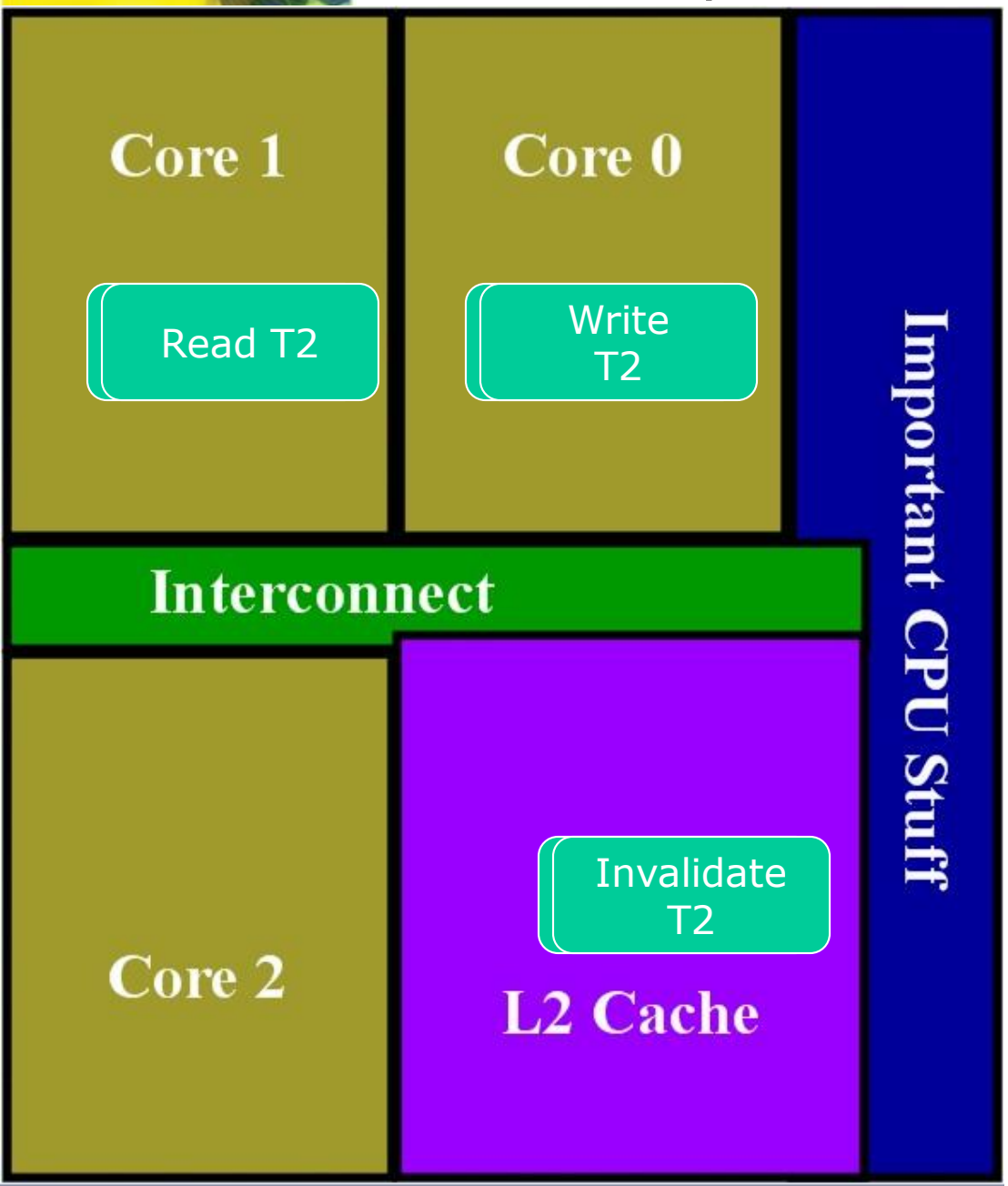
Dekker's/Peterson's Algorithm

```
int T1 = 0, T2 = 0;

Proc 1:
void LockForT1() {
    T1 = 1;
    if( T2 != 0 ) {
        …

Proc 2:
void LockForT2() {
    T2 = 1;
    if( T1 != 0 ) {
        …
}
```

Dekker's/Peterson's Animation



Proc 1:

Write T1

Read T2

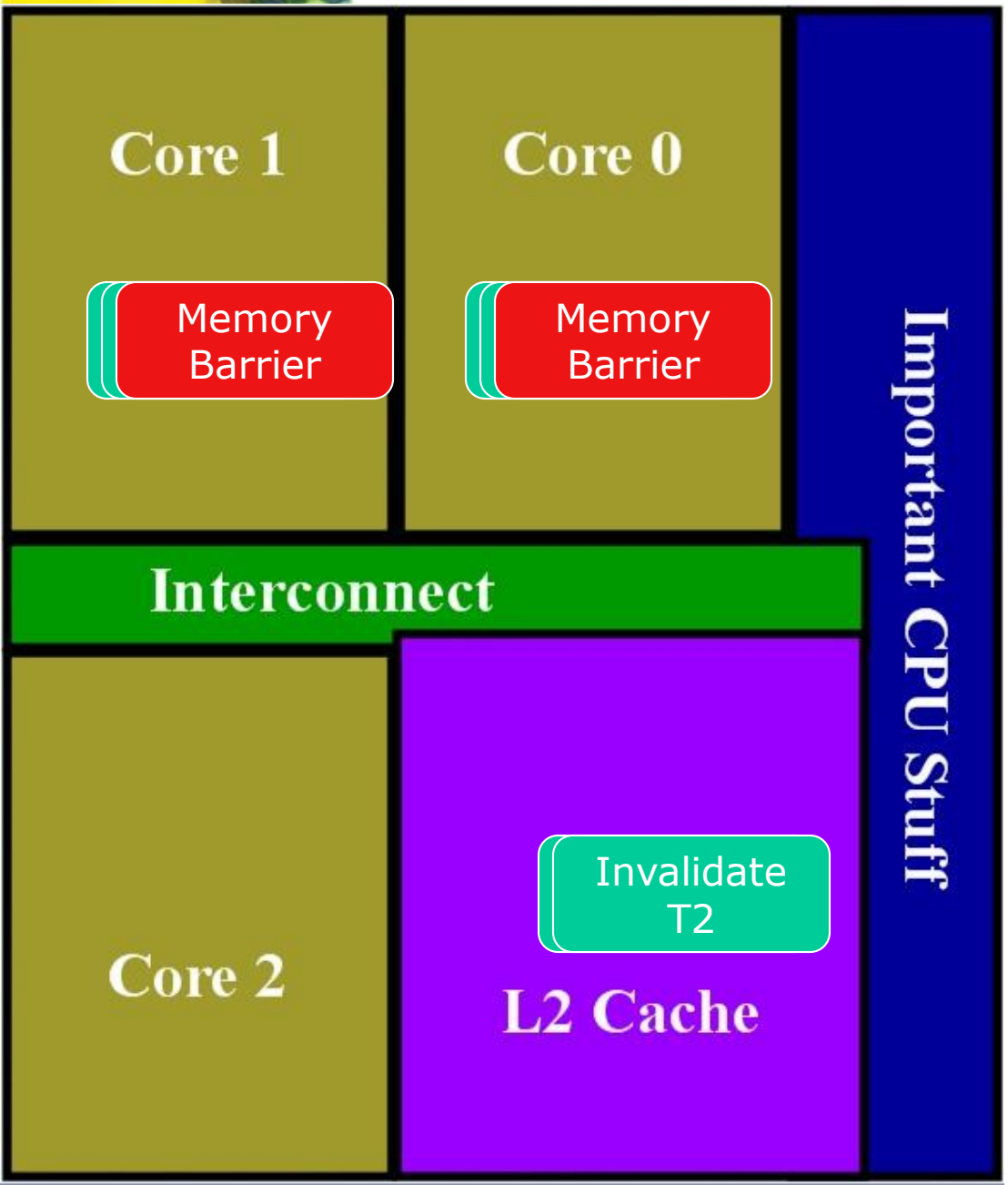
Proc 2:

Write T2

Read T1

- Epic fail! (on x86/x64 also)

Dekker's/Peterson's Animation



Proc 1:

Write T1

MemoryBarrier();

Read T2

Proc 2:

Write T2

MemoryBarrier();

Read T1

- It's all good!

Full Memory Barrier

- MemoryBarrier();
  - x86: __asm xchg Barrier, eax
  - x64: __faststorefence();
  - Xbox 360: __sync();
  - ARM: __dmb();
  - IA64: __mf();
- Needed for Dekker's algorithm, implementing locks, etc.
- Prevents all reordering – including preventing reads passing writes
- Most expensive barrier type

Dekker's/Peterson's Fixed

```
int T1 = 0, T2 = 0;

Proc 1:
void LockForT1() {
    T1 = 1;
    MemoryBarrier();
    if( T2 != 0 ) {
        …

Proc 2:
void LockForT2() {
    T2 = 1;
    MemoryBarrier();
    if( T1 != 0 ) {
        …
}
```

Dekker's/Peterson's Still Broken

```
int T1 = 0, T2 = 0;

Proc 1:
void LockForT1() {
    T1 = 1;
    MyExportBarrier();
    if( T2 != 0 ) {
        …

Proc 2:
void LockForT2() {
    T2 = 1;
    MyExportBarrier();
    if( T1 != 0 ) {
        …
}
```

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

Dekker's/Peterson's Still Broken

```
int T1 = 0, T2 = 0;

Proc 1:
void LockForT1() {
    T1 = 1;
    MyImportBarrier();
    if( T2 != 0 ) {
        …

Proc 2:
void LockForT2() {
    T2 = 1;
    MyImportBarrier();
    if( T1 != 0 ) {
        …
}
```

Dekker's/Peterson's Still Broken

```
int T1 = 0, T2 = 0;

Proc 1:
void LockForT1() {
    T1 = 1;
    MyExportBarrier(); MyImportBarrier();
    if( T2 != 0 ) {
        …

Proc 2:
void LockForT2() {
    T2 = 1;
    MyExportBarrier(); MyImportBarrier();
    if( T1 != 0 ) {
        …
}
```

THINK
GDC 09
learn network inspire
www.GDConf.com

What About Volatile?

- Standard volatile semantics not designed for multi-threading
  - Compiler can move normal reads/writes past volatile reads/writes
  - Also, doesn't prevent CPU reordering
- VC++ 2005+ volatile is better…
  - Acts as read-acquire/write-release on x86/x64 and Itanium
  - Doesn't prevent hardware reordering on Xbox 360
- Watch for atomic<T> in C++0x
  - Sequentially consistent by default but can choose from four memory models

Double Checked Locking

```
Foo* GetFoo() {
    static Foo* volatile s_pFoo;
    Foo* tmp = s_pFoo;
    if( !tmp ) {
        EnterCriticalSection( &initLock );
        tmp = s_pFoo; // Reload inside lock
        if( !tmp ) {
            tmp = new Foo();
            s_pFoo = tmp;
        }
        LeaveCriticalSection( &initLock );
    }
    return tmp; }
```

- This is broken on many systems

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

Possible Compiler Rewrite

```
Foo* GetFoo() {
    static Foo* volatile s_pFoo;
    Foo* tmp = s_pFoo;
    if( !tmp ) {
        EnterCriticalSection( &initLock );
        tmp = s_pFoo; // Reload inside lock
        if( !tmp ) {
            s_pFoo = (Foo*)new char[sizeof(Foo)];
            new(s_pFoo) Foo; tmp = s_pFoo;
        }
        LeaveCriticalSection( &initLock );
    }
    return tmp; }
```

Double Checked Locking

```
Foo* GetFoo() {
    static Foo* volatile s_pFoo;
    Foo* tmp = s_pFoo; MyImportBarrier();
    if( !tmp ) {
        EnterCriticalSection( &initLock );
        tmp = s_pFoo; // Reload inside lock
        if( !tmp ) {
            tmp = new Foo();
            MyExportBarrier(); s_pFoo = tmp;
        }
        LeaveCriticalSection( &initLock );
    }
    return tmp; }
```

- Fixed

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

InterlockedXxx

- Necessary to extend lockless algorithms to greater than two threads
  - A whole separate talk…
- InterlockedXxx is a full barrier on Windows for x86, x64, and Itanium
- Not a barrier at all on Xbox 360
  - Oops. Still atomic, just not a barrier
- InterlockedXxx Acquire and Release are portable across all platforms
  - Same guarantees everywhere
  - Safer than regular InterlockedXxx on Xbox 360
  - No difference on x86/x64
  - Recommended

Practical Lockless Uses

- Reference counts
- Setting a flag to tell a thread to exit
- Publisher/Subscriber with one reader and one writer – lockless pipe
- SLists
- XMCore on Xbox 360
- Double checked locking

Barrier Summary

- MyExportBarrier when publishing data, to prevent write reordering
- MyImportBarrier when acquiring data, to prevent read reordering
- MemoryBarrier to stop all reordering, including reads passing writes

- Identify where you are publishing/releasing and where you are subscribing/acquiring

Summary

- Prefer using locks – they are full barriers
  - Acquiring and releasing a lock is a memory barrier
- Use lockless only when costs of locks are shown to be too high
- Use pre-built lockless algorithms if possible
- Encapsulate lockless algorithms to make them safe to use
- Volatile is not a portable solution
- Remember that InterlockedXxx is a full barrier on Windows, but not on Xbox 360

THINK SERVICES
GDC 09
learn network inspire
www.GDConf.com

References

- Intel memory model documentation in Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide
  - http://download.intel.com/design/processor/manuals/253668.pdf
- AMD "Multiprocessor Memory Access Ordering"
  - http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf
- PPC memory model explanation
  - http://www.ibm.com/developerworks/eserver/articles/powerpc.html
- Lockless Programming Considerations for Xbox 360 and Microsoft Windows
  - http://msdn.microsoft.com/en-us/library/bb310595.aspx
- Perils of Double Checked Locking
  - http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- Java Memory Model Cookbook
  - http://g.oswego.edu/dl/jmm/cookbook.html

Questions?

- bdawson@microsoft.com

Feedback forms

- Please fill out feedback forms