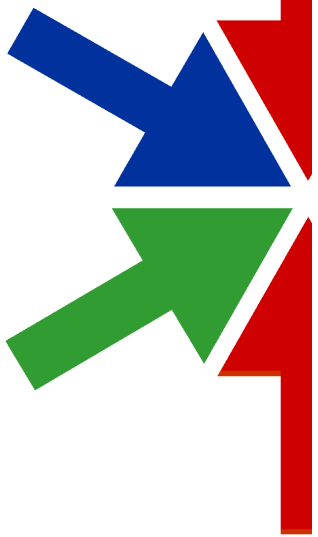
Understanding and Using SMP/Multi-Core Processors

New hardware and how to use it



Mike Anderson
Chief Scientist
The PTR Group, Inc.
http://www.theptrgroup.com

What We Will Talk About

- Some processor basics
- Parallelism in computing
- Motivations for multi-core processors
- Scaling issues
- O/S support for multi-processing
- Designing software for multi-processing
- Demo

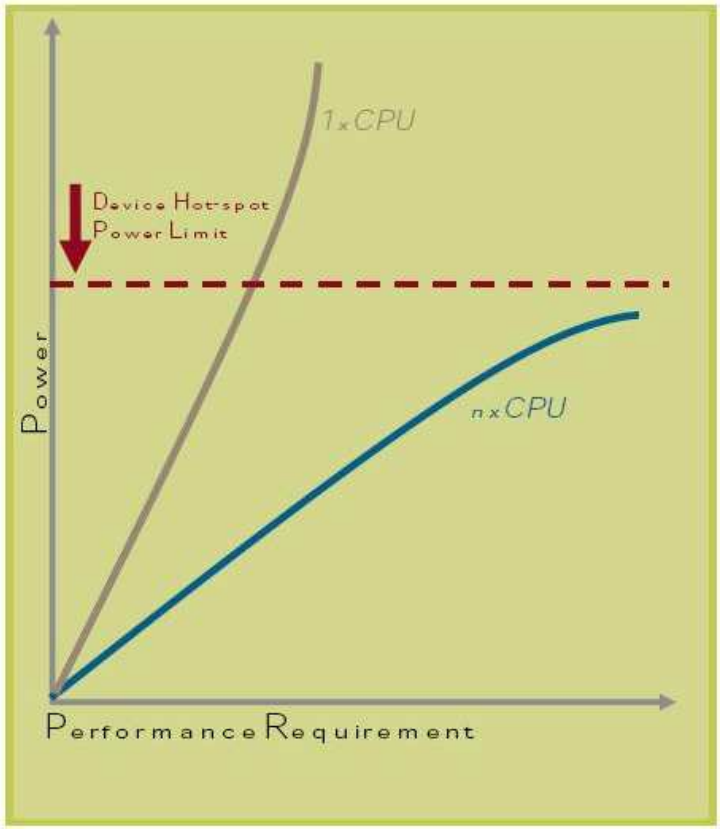
Boosting Performance

- Our industry is in a constant search for better performance
  - Better MIPS/watt ratios
- However, simply increasing clock frequencies has not been the panacea
  - Higher frequencies mean more voltage
    - Power consumption varies with the square of the voltage
- The option is to increase the number of instructions/second by adding parallelism

Scotty, We need more Power!

- Parallelism can be viewed at the micro (fine-grained) or macro (coarse-grained) level
  - Many processor families already support fine-grained parallel capabilities
- Largely due to power constraints, the semi manufacturers started moving to multiple cores
  - This has given rise to the current head-long rush to multi-core processing
- But, we're getting ahead of ourselves
  - Let's first examine some of the issues of processors in general

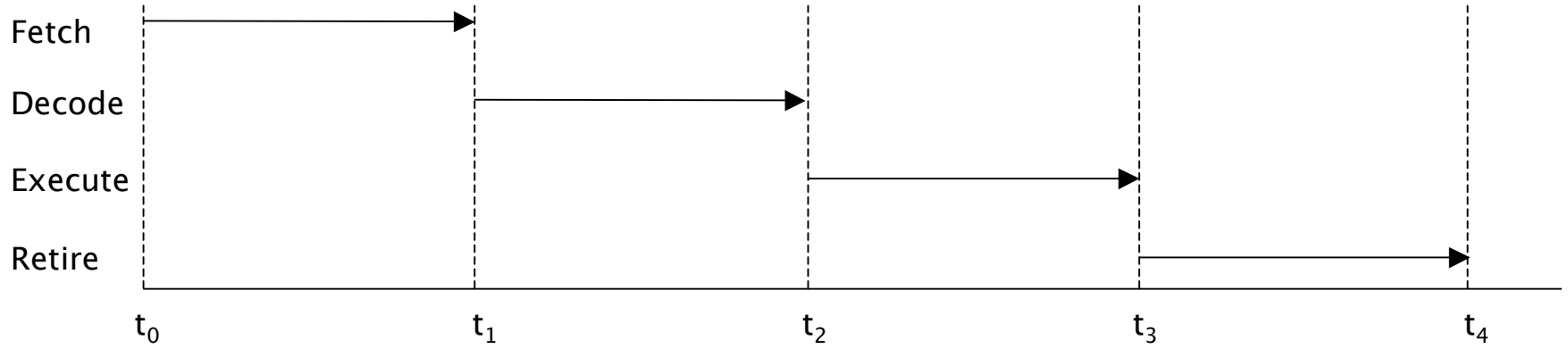
Source: Paramount



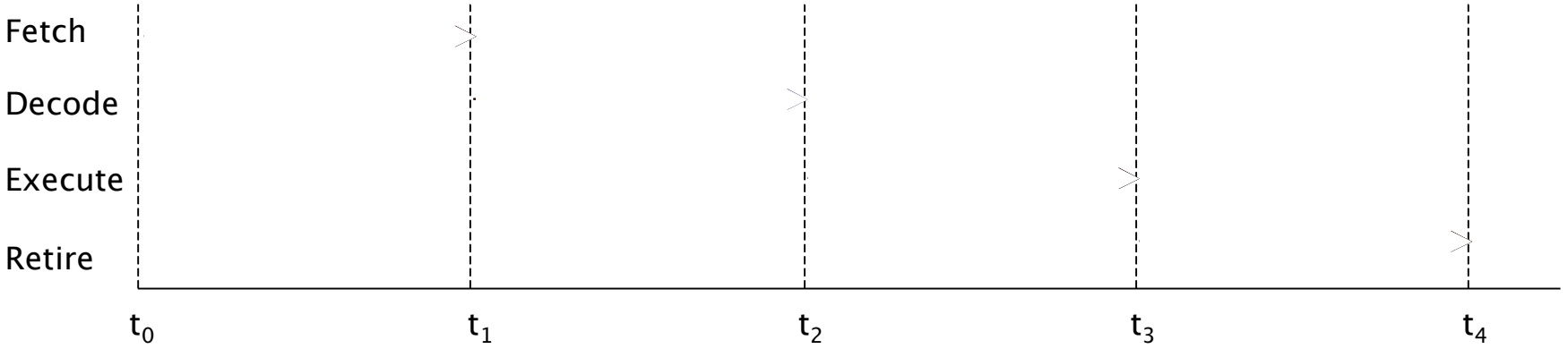
Source: FreeScale Semiconductor

More Speed Through Pipelining

- Given 4 cycles to execute an instruction:



- Use an instruction pipeline to make it faster



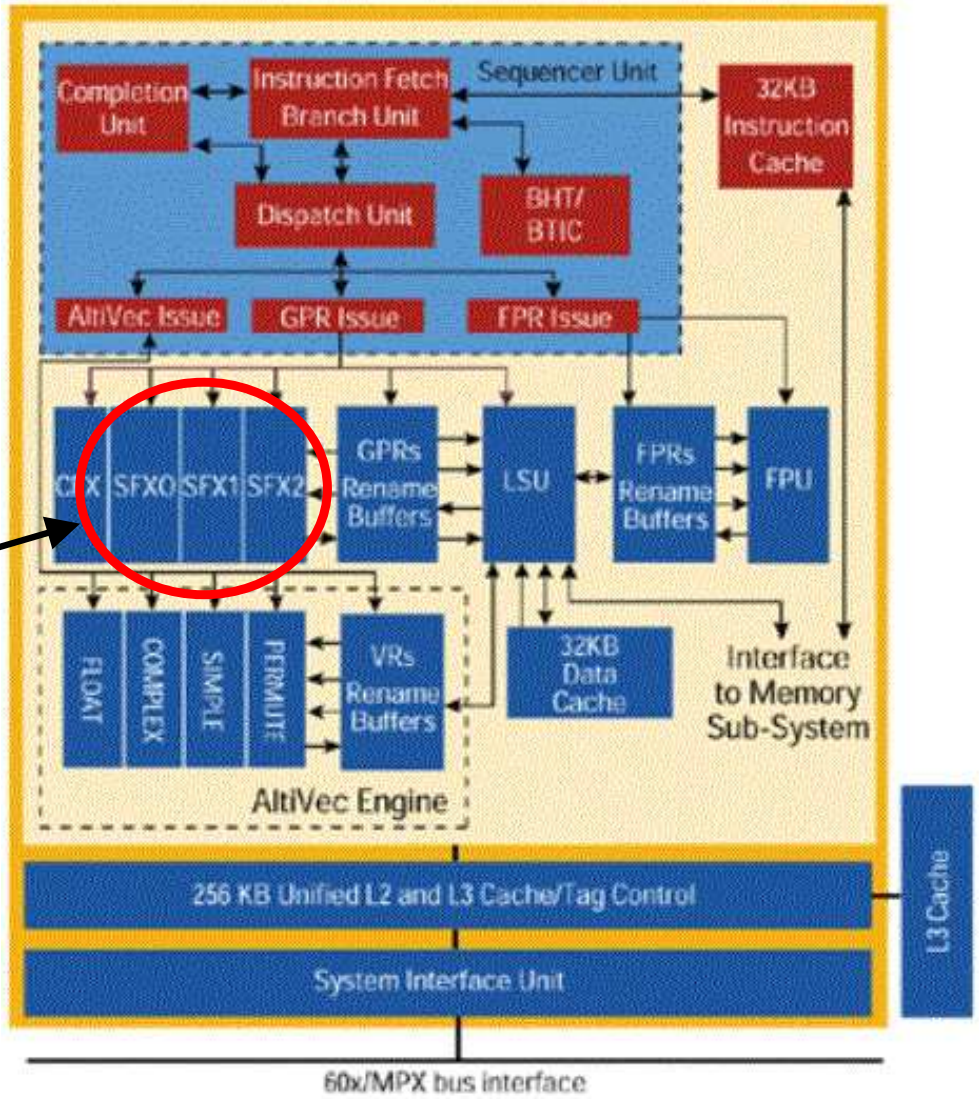
Classic vs. Newer CPUs

- The PPC 860, PPC 405 and 80486 are examples of scalar CPUs
  - Only one instruction/cycle
- With a properly tuned pre-fetch unit and associated pipeline, we can feed multiple instruction units per cycle
  - More than one instruction/cycle yields a super-scalar CPU
    - The Pentium, PPC 440, ARM 11, etc.
  - This is also called instruction-level parallelism (ILP)

Example Super-Scalar Processor

Multiple IUs

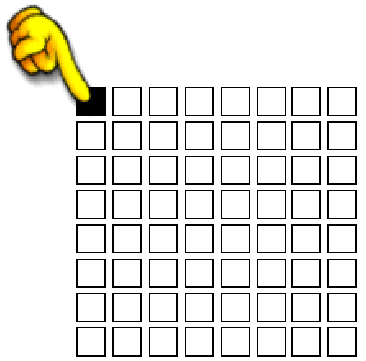
3 Instructions & 1 Branch per cycle on this processor!

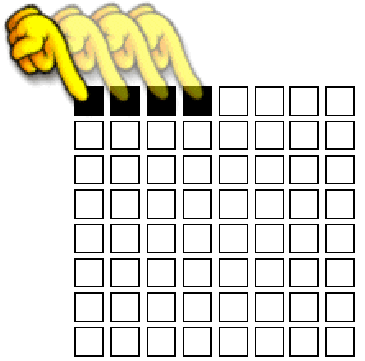


Source: Freescale Semiconductor

Fine-grained Parallelism via SIMD

- Some processors have specialized vector-processing hardware
  - IA SSE/MMX, Power Altivec, etc.
- Allows multiple operations on a matrix of data in a single command
  - Good for DSP, RADAR, graphics, etc.
- But, using SIMD instructions requires a lot of work on the part of the developer
  - They must understand the problem and map it to the vector units properly





Source: ARS Technica

AMP Clusters

- This is what we most commonly see in embedded/R-T systems
  - Multiple CPUs tied together via LANs or other interconnect strategy
    - Each CPU has its own copy of the operating system
  - E.g., media center extenders or Telco switching systems
- Non-Uniform Memory Architecture (NUMA)
  - Allocation of processes to processors is guided by the distance between the processor and the memory you want to use
- This is the realm of middleware
  - NDDS™, CORBA, SAForum's AIS
- The developer is responsible for allocating processes to processors
  - Each processor is referred to as a "node"

Cluster Characteristics

- Tasks are parceled out to processors as they become idle or they can be dedicated to a particular computation
- Communications latency becomes the key characteristic for use in real-time applications
  - Ethernets are cheap, but not deterministic
- They can combine load balancing and HA into a single system
  - Physical size is an issue

Source: Penguin Computing

Programming for AMP

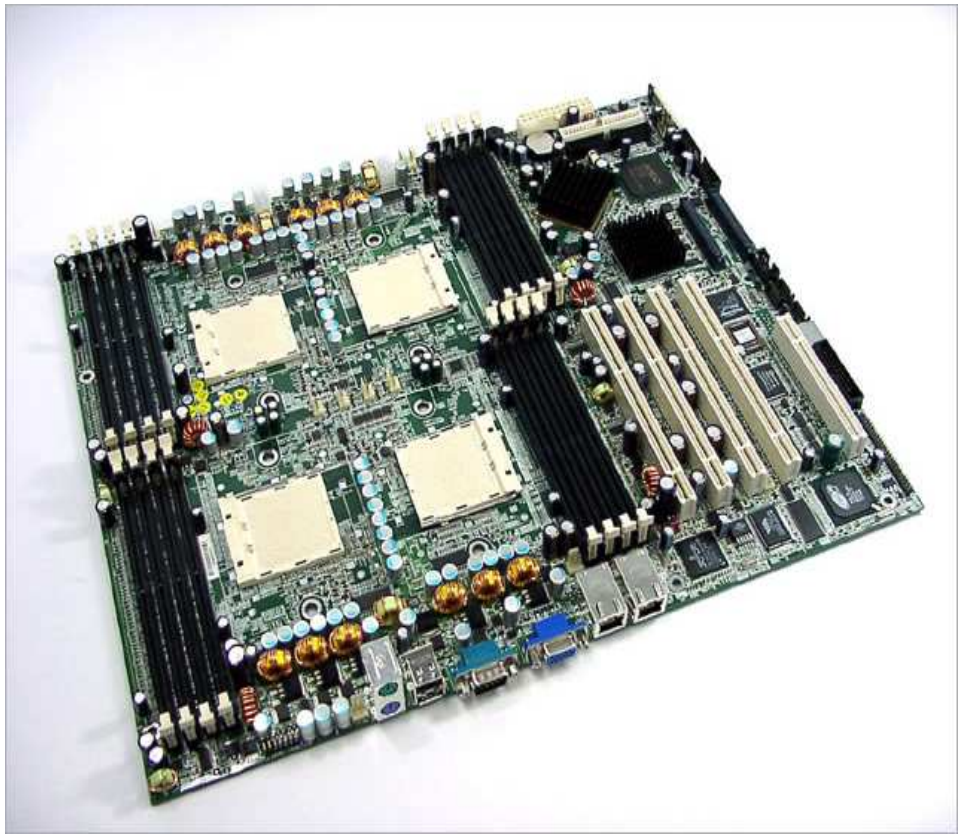
- No one standard is available for AMP
  - Multiple, competing APIs
- Requires the developer to decide which code needs to run where
  - You need to define message sources, sinks and distribution techniques
- Can be very complex, but performance of 100s-1000s of processors can be effectively harnessed for certain classes of problems

Characteristics of SMP Machines

- All processors see everything
  - Memory, I/O, interrupts, etc.
- There is only one kernel
  - The scheduler determines which applications are assigned to which processor
- Applications can migrate between processors
- They do not typically share caches
  - Although hybrid cache architectures are on the way

Example SMP Motherboard

- Quad-CPU AMD Opteron



Source: Tyan

Problems with SMP

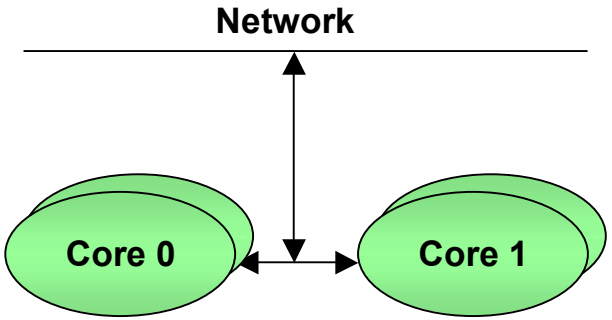
- SMP systems can be complex to set up and maintain because of duplicated hardware such as processor fans, etc.
  - They tend to be noisy as well due to fans
- SMP does not scale perfectly
  - Because the memory is shared between CPUs and the memory has a finite bandwidth, SMP machines can develop “hot spots” where multiple applications must serialize on a single piece of data
- Process migration can lead to poor cache utilization
  - We need to flush the caches if a process migrates
  - This also applies to ISR migration
- Multiple processors can lead to race conditions
  - We need to provide for multi-processor synchronization

Multi-Core Performance Issues

- Shrinking silicon processes allow for placing multiple, complete CPUs on the same die
  - However, synchronization and memory bandwidth issues limit our performance in bus-based interconnects
- Dual core runs at about 180% of single core of same speed
  - Quad core runs 50% faster than the dual
    - 270% faster than the single core
- Multi-core is typically clocked slower than a single core
  - Lower heat production and power consumption
  - But, poorer performance for single-threaded applications

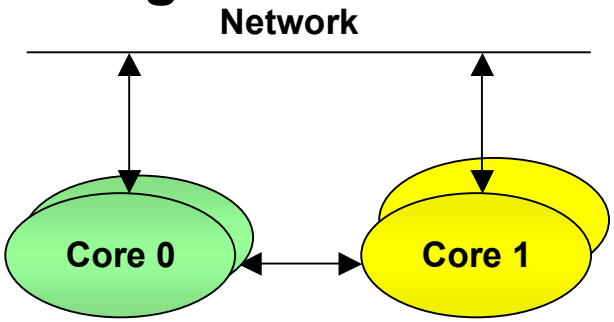
Multi-Processor Use Cases

SMP



- O/S manages applications transparently
- Good for control plane
- Bus bandwidth a limit for data plane

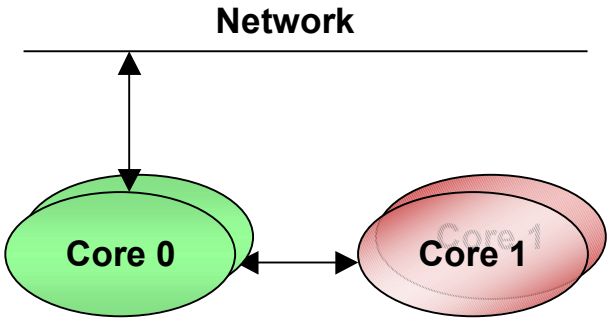
Partitioning



- Typically AMP
- Frequently implemented via light-weight executives or hypervisors
- Works for both control & data plane
- Partitioned processors can run alternate O/S or thin layers
- Partitioned processors are data shufflers
- Data plane cores can be simpler and cheaper
    - But, deep packet inspection suffers if they're too simple

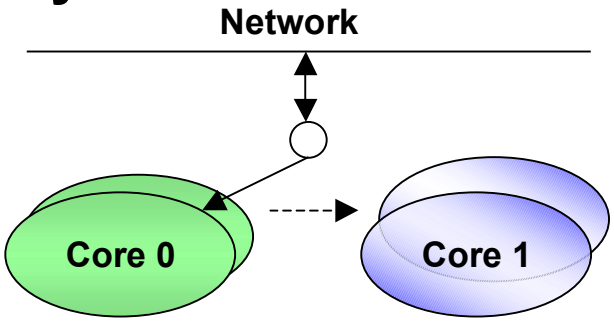
Multi-Processing Use Cases #2

Offloading



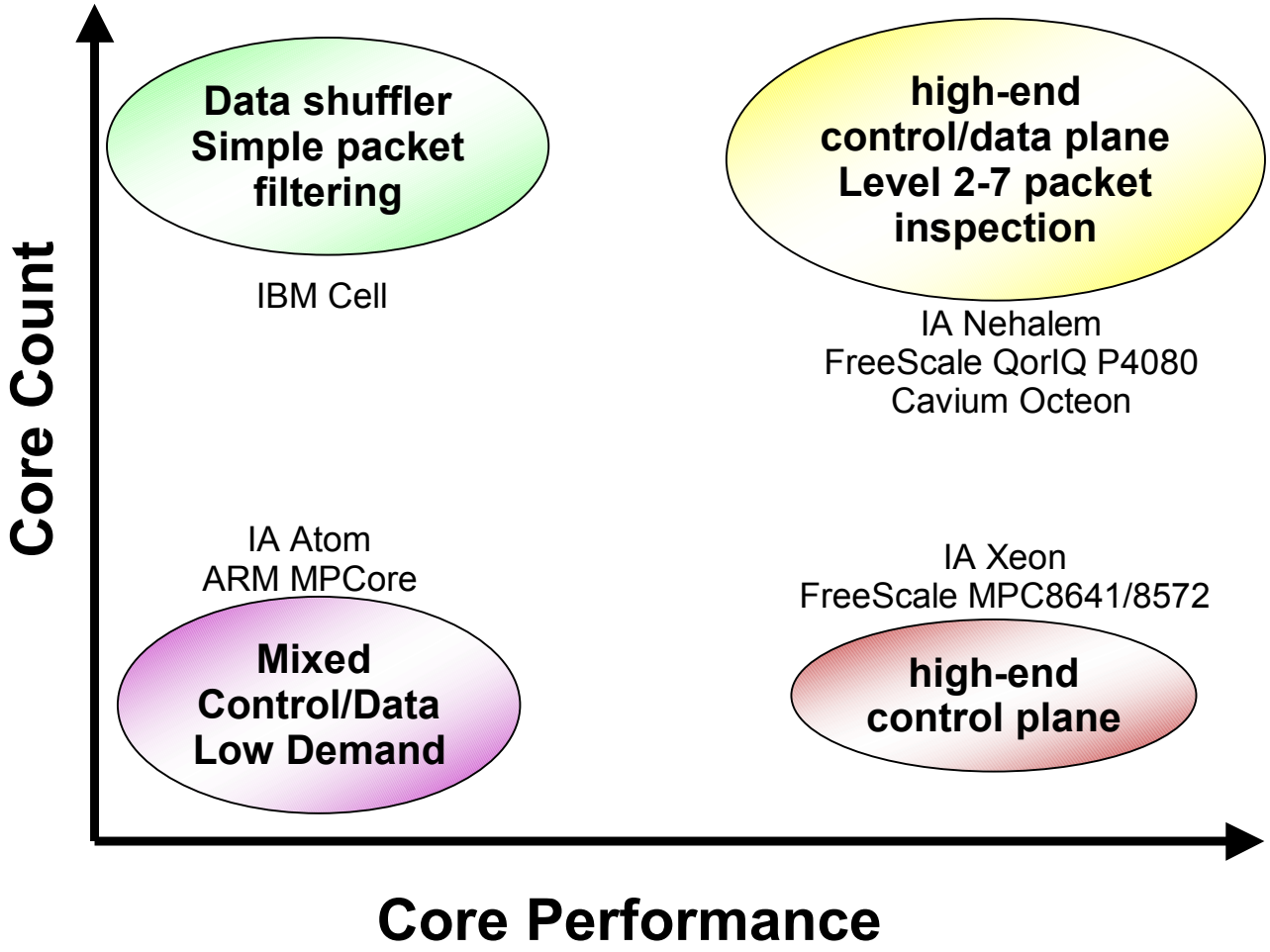
- CPU-intensive work is sent to alternate core(s) with thin executive
- Used in deep packet inspection and security applications

Standby



- Idle cores are held in reserve for redundancy
- Supports adding more capacity in the field via software
- Load updates to idle core and switch
  - Rapid S/W upgrade with little downtime

The Multi-Core Spectrum



Processor Affinity

- The term processor affinity relates to the tendency for an application to run on a particular processor and resist migration
- The scheduler will prefer not to migrate a process to another CPU unless needed
  - This is referred to as soft affinity
  - This can be overridden with hard affinity assignments in source code
- Hard affinity APIs allow the developer to make explicit assignments to a processor or a group of processors
  - You decide where your code runs by setting a CPU bit mask for each thread via calls like Linux's **`sched_setaffinity()`** and **`sched_getaffinity()`**

Example of Interrupt Load Balancing

```
mike@defiant:~> more /proc/interrupts
           CPU0       CPU1
  0:   21427467   21403917   IO-APIC-edge      timer
  1:      13217      14317   IO-APIC-edge      i8042
  8:          5          0   IO-APIC-edge      rtc
  9:          2          0   IO-APIC-fasteoi   acpi
 12:      17306      23786   IO-APIC-edge      i8042
 14:     205456     206781   IO-APIC-edge      libata
 15:     158807     158750   IO-APIC-edge      libata
 16:    2291422    2290748   IO-APIC-fasteoi   nvidia
 17:     405909     400991   IO-APIC-fasteoi   ipw3945, eth0
 18:          3          0   IO-APIC-fasteoi   ohci1394
 19:     168320     164332   IO-APIC-fasteoi   uhci_hcd:usb1, ehci_hcd:usb5
 20:    2174764    2176161   IO-APIC-fasteoi   uhci_hcd:usb2, HDA Intel
 21:          0          0   IO-APIC-fasteoi   uhci_hcd:usb3
 22:          0          0   IO-APIC-fasteoi   uhci_hcd:usb4
 23:          0          0   IO-APIC-fasteoi   sdhci:slot0
NMI:          0          0
LOC:   42831216   42830668
ERR:          0
MIS:          0
```

Interrupt Balancing

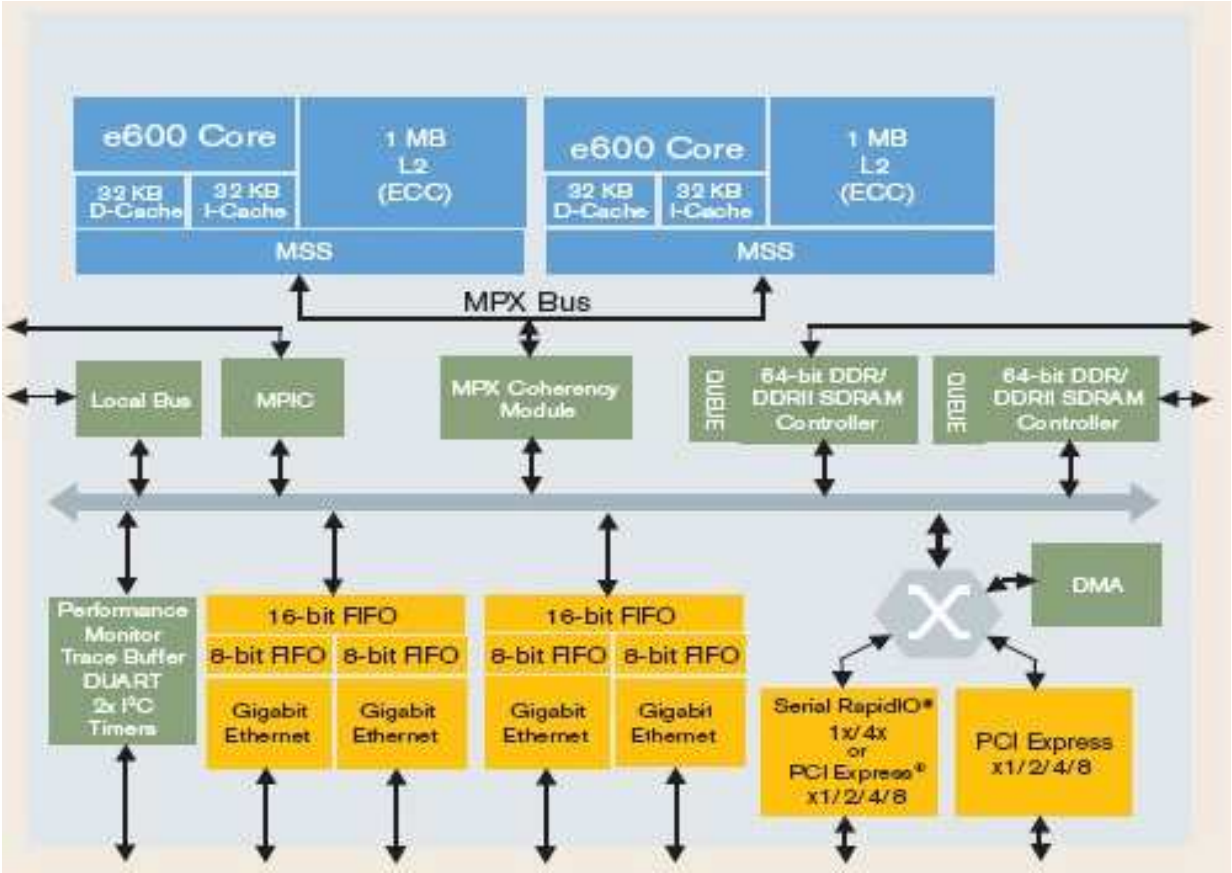
Interrupt Affinity

Non-X86 Multi-Core

- Intel and AMD are not the only silicon vendors with multi-core
- Each of the major core types have their own multi-core processors
  - ARM, MIPS, PPC all have multi-core implementations
- The following charts show just a few of the options

Dual-Core PowerPC from FreeScale

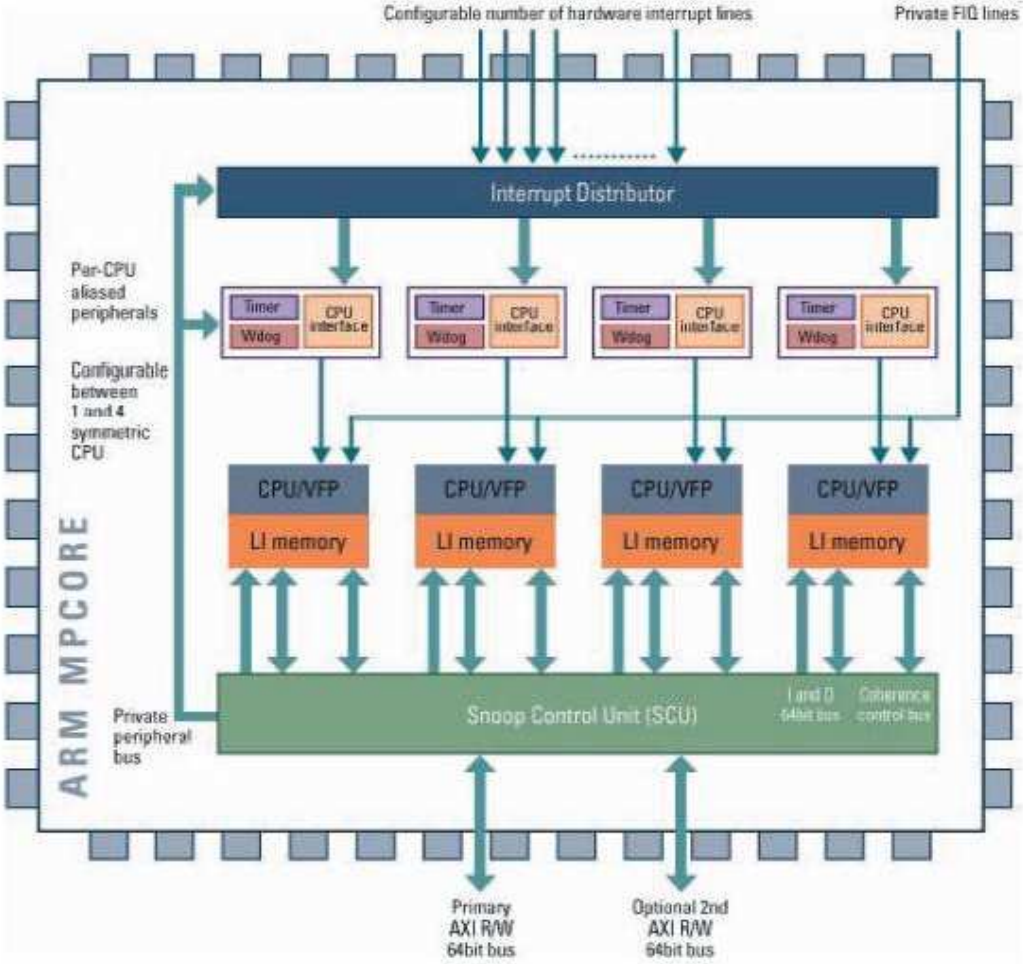
- MPC8641D
- Dual E600 cores can run SMP or detached mode
- MPX bus keeps the processor's caches coherent



Source: Freescale Semiconductor

Quad-Core MPCore ARM-11

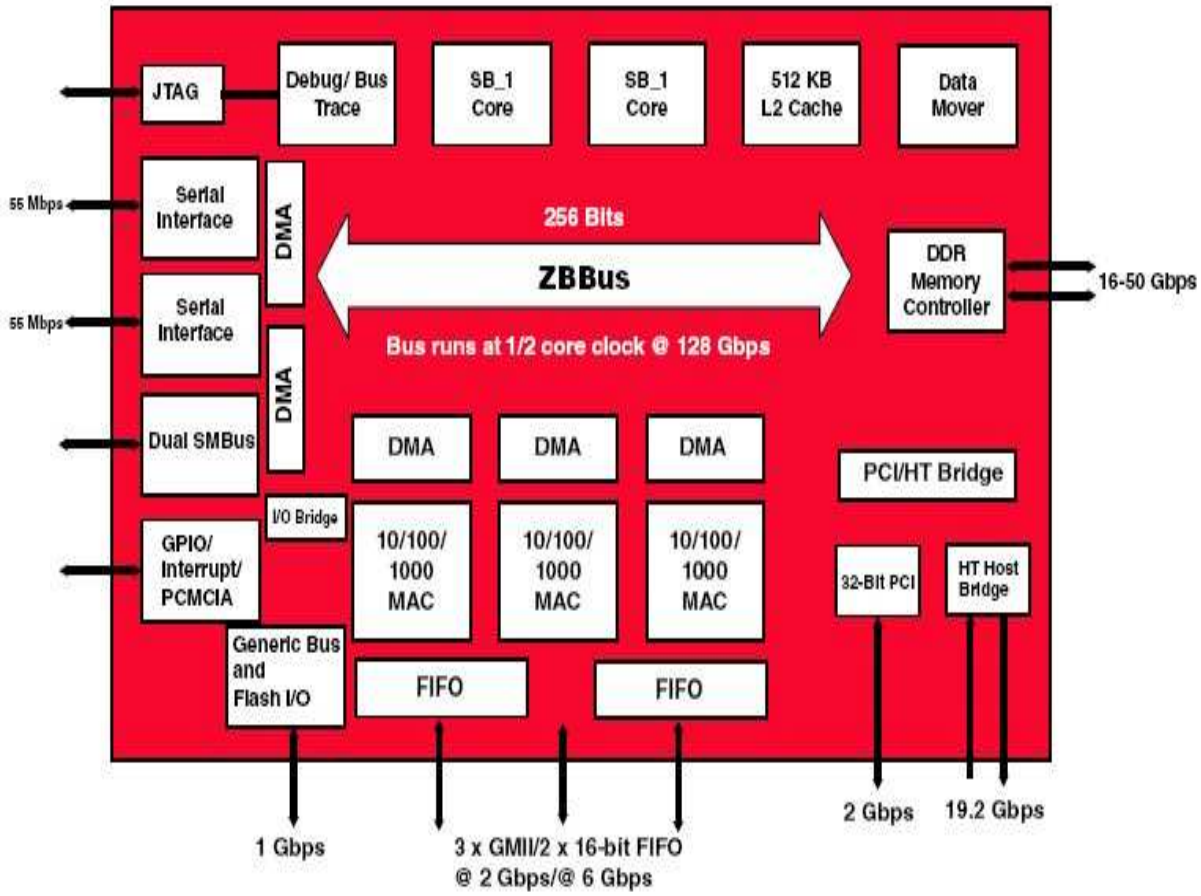
- Quad ARM-11 processors
- Specialized interrupt distribution for routing and interrupt balancing
- Bus snooping to improve cache coherency



Source:ARM Ltd

Dual-Core MIPS from Broadcom

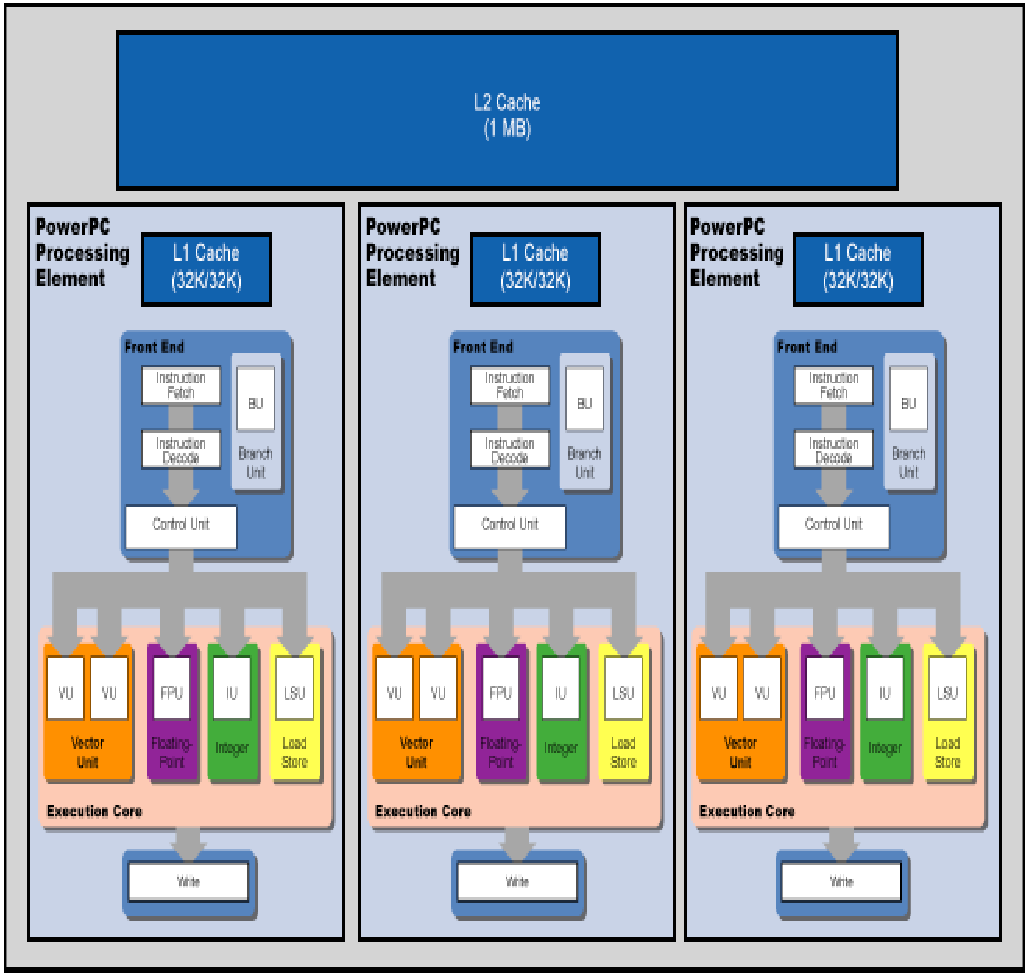
- Dual MIPS-64 with Quad-issue, in-order pipeline
- 600-800 MHz cores
- Power dissipation of 8-10W @ 800 MHz



Source:Broadcom Corporation

3 Cores and Beyond

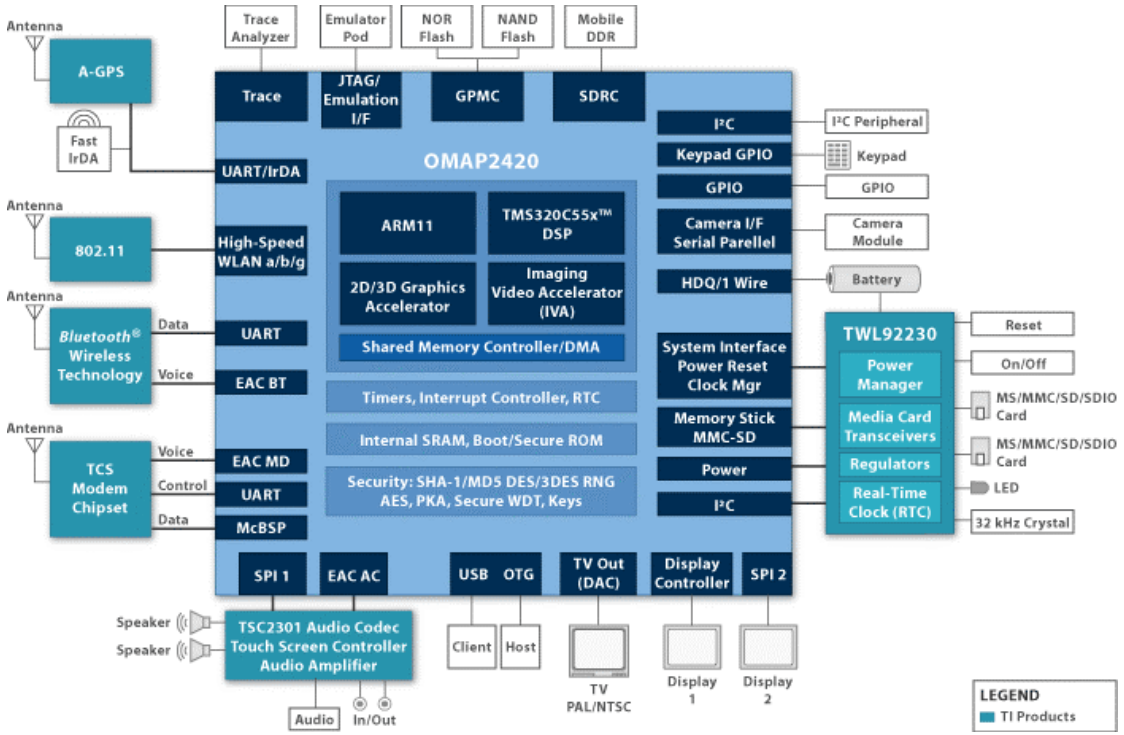
- IBM's Xenon™ is a 3-core, 64-bit PPC 970 used in the Xbox 360
- 8-16 cores are here or soon to be available
  - Cavium Octeon
  - FreeScale QorIQ P4080
  - Intel Nehalem



Source: ARS Technica

SSOCs in Embedded Systems

- The cellular phone marketplace is focused on providing a battery-operated, convergence device
  - Phone, PDA, game machine, camera, MP3 player, TV and more
- This requires components from multiple technologies



Source: TI

Source: PalmOne

Special-Purpose Heterogeneous MCPs

- The IBM Cell processor is another example of a heterogeneous multi-core processor
  - Built for the PS/3 game console
- But, it makes an excellent RADAR processing engine
  - High-performance computing engine

Source: Sony

The CELL Architecture

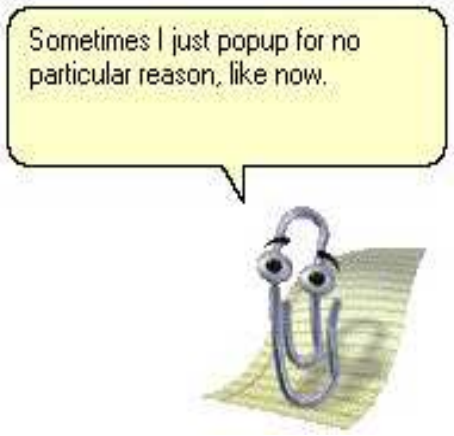
Source: IBM

Advantages/Disadvantages of Heterogeneous Cores

- The biggest advantage the ability to match the algorithm to the processor core
  - Traditional μP cores for user interfaces and control applications
  - DSP cores for signal processing and audio/video CODECs
  - Power consumption can be significantly lower in this approach
- However, you need two sets of build tools and special knowledge of the extra cores

Threading Example

- A reasonable example of processes vs. threads would be an application like MS Word
  - Word is the process that anchors the VMA
- Word is comprised of multiple threads
  - Repagination
  - Background printing
  - WYSIWYG formatting
  - Spell checking
  - Popping up that annoying paper clip thingy
  - And more…



Source: Microsoft

Scheduling Threads

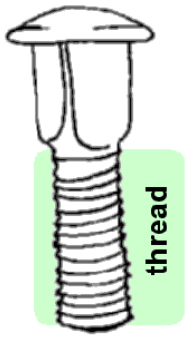
- There are several different approaches to dispatching threads in various O/Ses
- One approach sees only the processes
  - The threads then run a second-tier scheduler to see who gets the process' time slice
    - The M:N threading model of OS/X, Solaris and some Windows
- Another approach sees all threads as separate entities
  - The 1:1 threading model of most RTOSes and Linux
- Some allow a mixture of both

Threading APIs

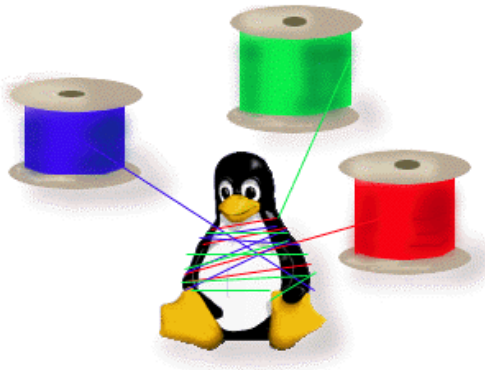
- Windows™, OS/X™ and Solaris™ all have their own threading APIs
  - These implement a M:N threading model in most cases
- Glibc has the GNU Pth API
  - Also an M:N threading model
- POSIX implements its own threading API known as pThreads
  - Works in either M:N or 1:1 models
  - Available on virtually all O/S platforms including Windows®, Linux, and many of the RTOSes
    - Windows® has both native and pThreads implementations

Confusion as to what a Thread is…

- Many developers are intimidated by threading in their applications
  - They are not quite sure what a thread is
  - O/S APIs can be difficult to understand
- Essentially, if you can think of a piece of code a separate sequence of steps from the main, then its probably a candidate to be a thread
  - A thread can be thought of as a subroutine with a life of its own



Source: myword.info



Source: acm.org

Fine-Grained Threading via OpenMP

- Open standard focused on extending compilers to support fine-grained parallelism via threading
  - Goal is high-performance by splitting up algorithms and running them as parallel threads
- Targeted at simultaneous multi-threading (SMT a.k.a. hyperthreaded) and multi-core CPUs
  - Compiler is responsible for creating parallel threads
  - Compilers require hints from the developer for what to parallelize
- http://www.openmp.org

OpenMP Usage

- To use OpenMP, you may need to restructure your code:

```
for (j=0; j < num_elements; j++) {
    my_array[j] = startval;
    startval++;
}
```

- This loop cannot be parallelized because of the data dependency on startval
  - We need to rewrite the code like this:

```
#pragma omp parallel for
for (j=0; j < num_elements; j++) {
      my_array[j] = startval + j;
}
startval += num_elements;
```

Programming for OpenMP

- OpenMP is only supported by certain compilers
  - E.g., Intel compilers for C/C++ and FORTRAN
  - GNU gcc 4.2.1+
- Requires the use of various `#pragma` directives to provide hints for the compiler
  - You need to know where they might apply
- May require you to recode your program to make it more parallelizable

Stepping up a Level – pThreads

- Of all of the threading APIs, the POSIX pThreads API has arguably the largest number of implementations
  - A non-proprietary API that can be implemented in virtually any O/S
- The threads all live in the global address space of the parent process VMA
  - Threads can each have their own priority
    - Different scheduling policies are also supported
- However, pThreads have a reputation for being difficult to understand

pThread Example #1 of 3

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

int global;

void * thread(void *joiner) {
    void *status;
    global = pthread_self();
    sleep(1);
    printf("Parent PID is %d, TID is %d, global = %d\n",
           getppid(), pthread_self(), global);
    if (joiner) {
        if (pthread_join((pthread_t)joiner, &status)) {
            exit(1);
        }
    }
    pthread_exit((void*) 0);
}
```

pThread Example #2 of 3

```
int main(void) {
    void            *status;
    int              x;
    pthread_attr_t   attr;
    pthread_t        curr_thr_id;
    pthread_t        prev_thr_id;

    pthread_attr_init(&attr);
    if (pthread_attr_setschedpolicy(&attr, SCHED_RR)) {
        exit(1);
    }

    /* Start 3 threads */
    prev_thr_id = 0;
    for (x=0; x<3; x++) {
        if (pthread_create(&curr_thr_id, &attr, thread, (void*)prev_thr_id)) {
            exit(1);
        }
        prev_thr_id = curr_thr_id;
    }
```

pThread Example #3 of 3

```
   /* Join last thread */
     pthread_join(curr_thr_id, &status);
}
```

- This example shows the same piece of code being used to create three different threads
  - Each thread is independent, but shares the VMA of main
  - Each could have its own priority and processor affinity assigned
  - In a 1:1 threading model, each would be independently schedulable

Reentrancy and Synchronization

- Thread APIs like POSIX support semaphores, mutexes, message queues and a host of other IPC mechanisms
  - Due to the flat address space within the VMA, critical sections need to be protected to avoid reentrancy issues
- Use of semaphores can enforce ordering of threads
  - Blocking one thread does not block all threads in the same process in 1:1 thread models

Simplifying Writing Thread Code

- Most threading APIs, although fairly straightforward, have been wrapped in class libraries
  - C++, Java, Python, Ruby, etc.
- Some, like Intel's Thread Building Blocks are open source and run in multiple O/Ses
  - http://osstbb.intel.com/
- Others are bundled into development tools such as Visual Studio
  - Use the approach that works for you

Migrating to Multi-Core

- If your applications is single-threaded, simply recompile for the platform and run
  - Don't be surprised if the performance actually drops from that of a single core due to clock-speed issues
- If the application is multi-threaded, try a containment approach first
  - Use affinity settings to lock the threads to a single core
  - Then start enhancing with mutual exclusion to enable threads running on multiple cores

Threading Design Guidelines

- When developing applications, try to identify those activities that can run in parallel
- Identify data flow through the application
  - Determine what data must be shared between activities
- Identify the correct sequencing of the activities
  - Temporal correctness
- Identify relative importance of activities
  - These may need priority adjustments

Thread Design Guidelines #2

- Don't assume that priorities will preclude race conditions
  - Lower priority thread can run on other core!
- When designing your threads, keep them as separate as possible
  - Don't share data unless necessary
  - Use synchronization primitives when needed
    - Semaphores, mutexes, message queues, etc.
- Try to keep data used by threads on separate cache lines
  - Create a cache_aligned_malloc/cache_aligned_free to make sure data is in separate cache lines to avoid false sharing
    - Avoid ping-ponging between processor caches

Thread Affinity Guidelines

- If your hardware is SMP/Multi-Core, run the application without adjusting the affinity to see if there is a problem
  - Don't try to solve a problem if it doesn't exist
- If there is an issue, look at processor loading to see if one processor is bearing most of the effort
  - If yes, then adjusting affinity comes next

Threads and Multi-Core/SMP Systems

- In O/Ses that support the 1:1 model, each thread will have its own CPU affinity settings
  - Individual threads could be assigned to different CPUs
- In M:1 or M:N threading models, the threads typically execute on the same processor as the main process
  - You can set the process affinity, but not the individual thread's affinity

Interrupt Affinity Guidelines

- Interrupt affinity can be tougher
- You'd like the ISR to always execute on the same core due to cache effects
- But, having all of the interrupts on the same core can cause poor interrupt latency
  - Make assignments carefully
- Frequently, the only way to determine the optimal assignment is to test, test, test

Example: Vehicle Telematics

- Dual-core ARM running Linux
  - Engine monitoring, GPS, MP3, and DVD playback functions
- Identify the “important” applications
  - Engine monitoring (gas, oil, water, ABS, engine operation, speed, temperature)
- Identify the low-latency applications
  - DVD, MP3

Example: #2 of 7

- Without the important applications, the car is undriveable
  - We could collect them into a single process with threads running at different update frequencies
    - Are the devices interrupt driven or polled?
    - If interrupt driven, these might be able to be handled via a set of kernel threads or within the ISR themselves
    - If polled, user threads can use nanosleep to wake up periodically to poll the sensors

Example: #3 of 7

- The low-latency applications require a guaranteed service level
  - 24 Hz for the video
    - NTSC is 30 FPS, PAL is 25 FPS
  - Audio sample rates can be high, but the jitter is typically less than 5ms
    - 250 Hz is typically fast enough to minimize jitter for MPEG audio
- This might require affinity assignments, but we'll test it first

Example: #4 of 7

- DVD playback lends itself to threading
  - Threads for each of:
    - Read the media, video CODEC, audio CODEC, user interaction (UI), render video, playback audio
- MP3 is similar to DVD without video
- GPS is a separate process
  - Threads for reading GPS radio, read map data, render map with location, UI
  - Frequency for update can be 1-2 Hz

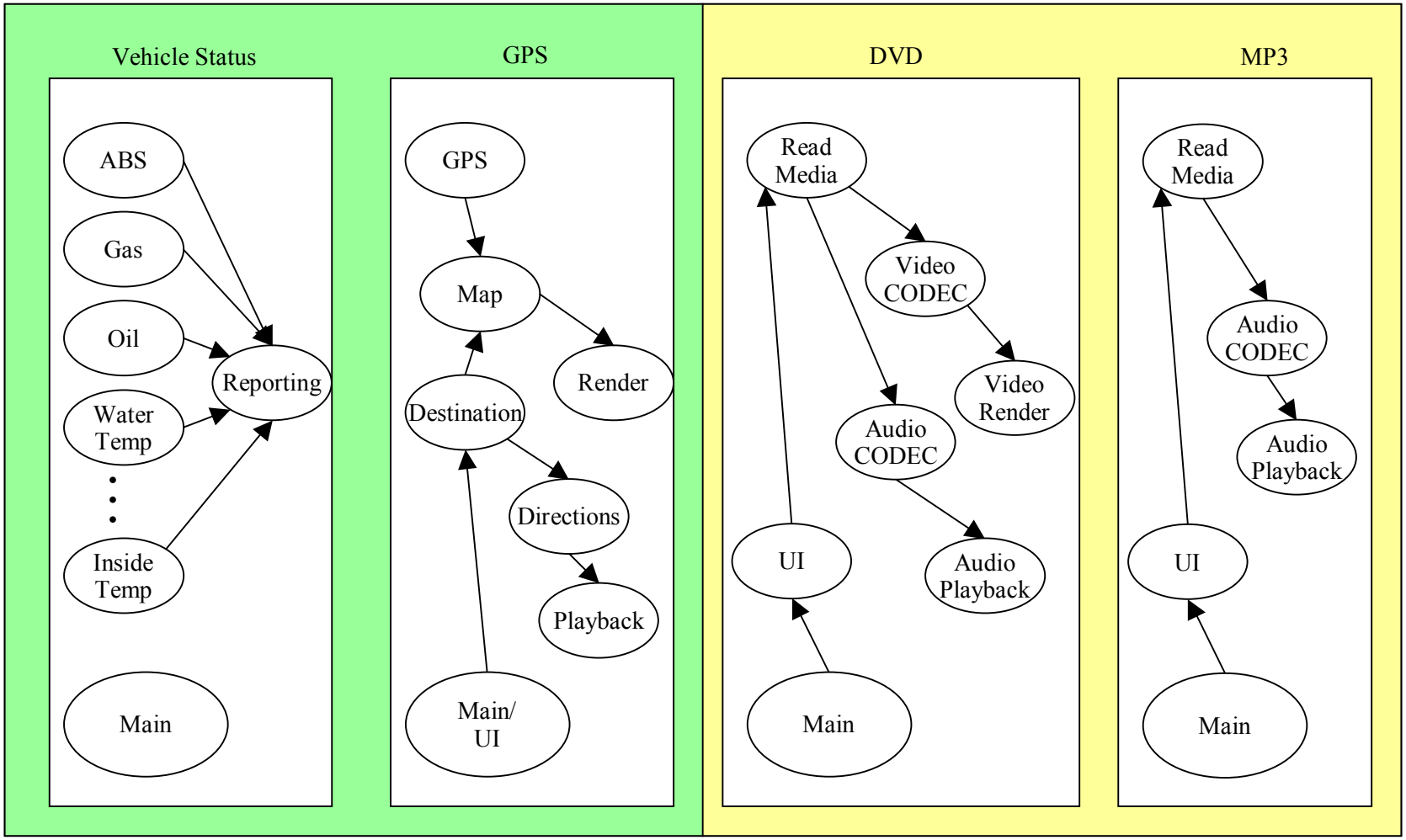
Example: #5 of 7

- Based on frequency, GPS & vehicle health/status could both run on single core
  - DVD/MP3 probably needs to be on a separate core unless we have hardware assist on CODECs
- Based on what we have so far:
  - 4 processes, 15-20 threads
  - Priority assignments and affinity still need to be tested

Example: #6 of 7

- This example was not exhaustive
  - Designed to give you a feel for the process
- Detailed design for each of the individual processes would follow
  - Must develop thread synchronization strategy
  - Number of threads will probably change during design

Example: #7 of 7



No One Answer

- It is unlikely that there would ever be just one way to solve a problem
  - Assignment and number of processes and threads is based on experience and experimentation
- There are many alternative ways to achieve the same goals
  - Someone just has to step up to the plate and pick one
  - Don't be afraid to change course if something isn't working

Summary

- SMP systems have been with us for quite some time
- Multi-core is just SMP on a chip
- Make sure your O/S supports affinity mechanisms
  - Provides the most flexibility
  - Availability of priorities is nice too
- We must consider application redesign to take advantage of multi-core processors
  - The use of threads becomes important
  - POSIX pThreads API exists on most O/Ses
    - Good documentation, good place to start