Challenges in programming multiprocessor platforms

John Goodacre
ARM Ltd

MPSoc’04
4th International Seminar on Application-Specific Multi-Processor SoC

5 - 9 July 2004
Hôtellerie du Couvent Royal, Saint-Maximin la Sainte Baume, France
First, Some Terminology

- **Disclaimer** – I don’t have enough space or time to offer a definitive list of all MPSoC architectures….
  - So I’ll concentrate on MP in open platforms

- **Hardware processor arrangements**
  - Heterogeneous – multiple different processors
  - Homogeneous – multiples of the same processor

- **Software arrangements**
  - Asymmetric – running different code base
  - Symmetric – running the same code

- **Units of work**
  - Application – the problem to be solved
    - Defined by product requirements
  - Task – programmer bounded representation of work within an application
    - Defined at design time
  - Thread – a mechanism to implement tasks within an application
    - Used during software implementation
Challenges in Software Design

- **Time schedules**
  - It’s only software, you can make it do anything…

- **Programmability is essential**
  - Increasing complexity when running multiple dynamic applications
  - Tools / visibility of software is getting harder
  - Verification / repeatability
  - Design and test development environments are getting more complex

- **Reusability is a fact of life!**
  - Needing portability of solution
    - Needing a layered abstraction of functionality
Hardware is reaching physical limits

- Classical single instruction context (uniprocessors) are failing to scale using current methods
  - Can’t extract more from instruction level parallelism

- Processor engines are needing to get help from the application programmer
  - Getting developers to represent their application using multiple instruction context

- High performance from high MHz is reaching thermal / energy limits in desktop and embedded
  - “Intel cancels P4 in favour of multicore”
  - “ARM announces multiprocessor core”
  - “IBM says scaling from process reduction is dead”
Microarchitectures must evolve

Comparisons from public information. All processors using 130nm process.

Higher frequency cores use more power as voltage factor is squared
Power = k * MHz * vt * vt

MIPS 20Kc
- 1320 DMIPS
- MIPS 20Kc is 20mm² (32/32K cache)
- MPCore is 16mm² (16/16k x 2)

MPCore 2-way
- 20% Smaller

MPCore

MP7II-Mobile
- 2600 DMIPS
- PIll-M is 80mm² (32/32k + 256)
- MPCore is 37mm² (16/64k x 4)

MPCore 4-way
- 53% Smaller
‘Classic’ Heterogeneous Asymmetric

- T.I. OMAP ‘Dual-Core’ Applications Processor
  - ARM Host processor
  - T.I. Media DSP
Homogeneous Asymmetric

- For example: Network Processor
  - ARM PrimeXsys™ Dual-Core Platform
- Channel Processing
  - NAT, Firewall, IP stack
- Host Processor
  - GUI and configuration
  - Email services
Asymmetric Multiprocessing (AMP)

- Software model that enables programmer to run multiple simultaneous applications
  - Uses a message based interconnect between both heterogeneous and homogeneous processors
  - Offered in various form (for a long time!)
    - Inmos Transputer (homogeneous MP)
    - Tensilica “sea of processors”
    - Custom designs, eg Agere eight way ARM966E-S™

- Provides an efficient solution when the application can be statically partitioned across processors
  - Allows effects of a task to be isolated from others
  - Provides a simple mechanism to grow existing code on to a MPSoC
Example of AMP code

**Application on host CPU**
- Prepares work
- Sends to slave CPU
- Waits for it to be done
  - Get on with something else
- Uses the work

**Application on slave CPU**
- Waits for work from host
- Does the work
- Send it back to host

```cpp
main() {
    while( ! Shutdown ) {
        work = GetWork();
        SendToWorker(work);
        work = WaitForWorkComplete();
        DisplayWork(work);
    }
}
```

```cpp
main() {
    while( ! Shutdown ) {
        work = ReceiveWork();
        DoWork(work);
        PostWork(workQueue, work);
    }
}
```
Challenges of Asymmetric MP...

- Programmer needs to split application and statically allocate sub-applications to processors
  - Possibly across different microarchitectures
  - Very difficult if you don’t ‘know’ the application

- The complexity of managing the dynamic workloads of open platforms breaks this model
  - Difficult to ensure efficient utilization of processors
    - Dynamic nature can overload specific processors
    - Difficult to provide single task scalability
  - All vendor solutions are different
    - Causing fragmentation in tools support
    - Need a rewrite / rearchitecture if you need to change
Symmetric Multiprocessing (SMP)

- Software model that enables programmer to utilize multiple instruction context architectures
  - Assumes common memory, common peripheral
  - Offered by various hardware architectures
    • Asymmetric MP with coherent interconnect
    • Symmetric MP with coherent caches
    • Multi-threaded uniprocessors with common cache

- Provides a common model to increase standards
  - Programmer uses threads to represent their tasks
  - Operating system schedule threads over processors
  - Seen as the next dominant programming model
    • Still portable between uniprocessor designs
Example of multi-tasked application

‘Typical whiteboard design’ of a video-phone
  – Application is initially designed as multiple tasks

Network Interface
  → Stream Processing
  ↓
  → Video Encode
  → Video Decode
  → Audio Encode
  → Audio Decode

→ Camera
→ Screen
→ Microphone
→ Speaker
Various implementation options

- **Uniprocessor**
  - Event driven, cooperative time sliced
    - Asynchronous work dispatch
  - Pre-emptive time sliced multi-threading

- **Multiprocessor**
  - Same as uniprocessor
  - With the OS also able to share threads over CPU
    - Reduces cost of context switching
    - Improves system level response

- Easiest in both cases is to simply map application tasks to threads
  - Allows existing code implementations to be used
Multi-threading mechanisms

- Fork-Exec: Create a thread on demand
  - Task has a clear start and end condition
  - The task is long lived
    • Enough to hide the cost of creating/killing thread
  - Useful to migrate existing code to multi-tasked app.
  - Each task likely to have multiple synchronization points
    • Incorrect partitioning can destroy performance

- Worker Pool: hand off work of to pool of workers
  - Application has clearly defined ‘units of work’
  - Pool of tasks waiting for work
  - Task synchronization best limited to split/merge of work unit
    • Need to ensure work items are not serially dependant
Example of multitasking

- **Fork-Exec**

  ```c
  main() {
    while( ! Shutdown ) {
      work = WaitForWork();
      CreateThread(WorkerTask, work);
    }
  }

  WorkerTask(work) {
    DoWork(work);
  }
  ```

- **Worker Pooling**

  ```c
  main() {
    for(i = 0; i < numCPU * 2; i++) {
      CreateThread(WorkerTask, workQueue);
    }
    while( ! Shutdown ) {
      work = WaitForWork();
      PostWork(workQueue, work);
    }
  }

  WorkerTask(workQueue) {
    while( ! Shutdown ) {
      work = WaitForWork(workQueue);
      DoWork(work);
    }
  }
  ```
Multi-tasked application using threads

Example implementation of the video-phone

```c
videophone()
{
    Struct {
        ...;
    } commonState;

    CreateThread(NetworkHandler, commonState);
    CreateThread(VideoEncoder, commonState);
    CreateThread(VideoDecoder, commonState);
    CreateThread(AudioEncoder, commonState);
    CreateThread(AudioDecoder, commonState);

    while( ! Shutdown ) {
        ProcessesStreams(commonState);
    }

    KillThreads();
}
```
Single task parallelisation

- Multi-tasking works well until a single task needs more performance than a single processor
  - Not a significant issue if the task is easily represented by multiple sub-tasks – eg CODEC
  - Sub-tasking can be complicated when:
    - Represented by a single linear algorithm
      - Especially if already set in code!
    - Algorithm is a sequence of inter dependent operations

- Luckily, looking for parallelism at the software code block or loop level can simplify these issues
  - Splitting iterations of a loop across processors
  - Placing separate sections of code on processors
Representing sub-tasks using OpenMP

```
// Multiply rows of A with vectors of b(i), and stores summed product in c(i)
float multiply_matrix(float A[][], b[], c[])
{
    int i, j;

    /* Use OpenMP’s managed pool of threads with scoped variables */
    #pragma omp parallel shared(A,b,c,total) private(i)
    {
        /* Loop work-sharing construct - distribute rows of matrix */
        #pragma omp for private(j)
        for (i=0; i < SIZE; i++)
        {
            for (j=0; j < SIZE; j++)
            {
                c[i] += (A[i][j] * b[i]);
            }
        }

        /* Update of running total must be serialized */
        #pragma omp critical
        {
            total = total + c[i];
        }
    }
    // end of parallel i loop
}
// end of parallel construct

return(total); // Matrix-vector total - sum of all c[]
```
Parallelisation of media codec tasks

- Effort only required when task needs more performance than a single processor can provide
- Example: MPEG2 decoder
  - Sampled from the ARM SMP Evaluation Platform
  - Demonstrates utilization of addition processors
Challenges of Symmetric MP…

- Difficult to isolate effects of a task from other tasks
  - All tasks share the same processors
  - OS API often provide affinity and task level prioritization
- Programmer needs to take care not to abuse common memory system
  - By requiring too many synchronization points
  - By causing data to need to migrate continually between processors
- Hardware must address processing sub-system bottlenecks
  - Around ensuring memory coherency
  - Around synchronization between processors
ARM MPCore Hybrid Multiprocessor

- **Snoop Control Unit (SCU)**
- **Interrupt Distributor**
- **Per-CPU aliased peripherals**
- **Configurable between 1 and 4 Symmetric CPU**
- **Private Peripheral Bus**

**Support for both AMP and SMP workloads**

- **Primary AXI R/W 64-bit bus**
- **Optional 2nd AXI R/W 64-bit bus**
- **Vector Floating Point (VFP) is optional**

**ARM**

**THE ARCHITECTURE FOR THE DIGITAL WORLD**

21
Conclusions

- Embedded open platform MPSoC can’t simply copy desktop microarchitectures
  - Can’t afford the cost of traditional coherency
    • Slow system bus used for snooping
    • SoC components used for communications
  - Needs to put low power consumption above peak performance

- Solution must be programmable by the “open platform” development team
  - Allowing migration of existing code base
  - Using a ‘mass-market’ model that can also offer high levels of processing efficiency