ARM 指令集体系结构下的并行机制

作者：
John Goodacre
Andrew N. Sloss
ARM

通过不同层次的并行处理，基于ARM内核的芯片将逐渐改变人们对技术的使用方式。随着ARM芯片销售量的迅猛增长（现在一年有15亿颗以上的ARM芯片的销售量），程序员现在可以为他们编写的代码找到更加广阔的市场空间。

在过去的15年里面，ARM的精简指令集（RISC）处理器不断发展演化，已经形成了一个丰富的产品系列，目前最高端的产品为成熟的多处理器内核。嵌入式应用对性能需求的不断增长和新技术对效率的提高一直在推动着ARM体系结构的进化。

在ARM体系结构的发展过程中，ARM的开发团队使用了计算机架构下的各种技术来提高并行性。ARM用来改进性能和效率的技术包括可变执行时间、子字并行化、类数字信号处理的操作\线程级别的并行化和异常处理，以及多处理器内核架构。

纵览ARM体系结构的发展历史，我们可以看到不同阶段的ARM处理器采用了不同的并行处理器技术。目前，并行程度最高的是ARM11 MPCore多处理器内核。

针对嵌入式应用的RISC处理器

早期的RISC设计，例如MIPS，完全的将注意力集中在高性能上。设计者通过相对较大的寄存器组、数目精简的指令数目、load-store架构、以及相对简单的流水线实现了这个目标。这些现今已经被普遍认可的方法在很多现代处理器设计中都能够找到。

ARM的RISC处理器与之相比有着很多不同的特点，部分原因是ARM处理器是作为SoC中的嵌入式内核设计的。尽管设计的主要考虑因素仍然是性能，但是其他方面，设计者也给予了充分考虑，例如高代码密度、低功耗、小硅片面积等重要因素。

为了达到这些设计目标，ARM的研发团队将RISC的规则做了一些改进，如引入了执行周期可变的指令，增加了内插式桶形移位器（inline barrel shifter）对一个输入寄存器进行预处理，条件执行，压缩的16位THUMB指令集，以及一些增强型的DSP指令。

- **可变执行周期：**基于load-store架构，ARM处理器在处理数据之前必须首先把数据加载到一个通用寄存器中。如果按照原始的RISC设计中的单周期限制，单独地加载和保存每个寄存器并不是一种高效率的方法。为此，ARM处理器增加了可以同时加载或保存多个寄存器的指令。这些指令根据需要进行数据传输的寄存器数目不同，执行需要的时钟周期数目也不同。这项技术对进入和退出子例程时保存和恢复现场非常有用。它可以改善代码密度，减少预取的指令，减少系统的功耗。

- **内插式桶形移位器：**为了使数据处理指令更加灵活，在对寄存器处理之前可以先对其进行移位或者循环操作，这项技术给了每个数据处理指令更多的灵活性。

- **条件执行：**ARM指令可以仅在满足特定条件时才执行。指令的执行条件一般通过指令助记符的后缀标

记。举例来说，对最大公约数算法使用条件执行指令，比不使用条件执行指令可以节省12个字节，大约节省了42%的代码空间。

- **16位的THUMB指令集：**ARM指令集的16位压缩版本可以以轻微的性能损失换来很高的代码密度。因为16位的Thumb指令集是为编译器而设计的，它并不支持对ARM32位指令架构下的全部寄存器的访问。使用Thumb指令集可以显著降低程序的大小。

2003年，ARM宣布了Thumb-2技术，它可以进一步改进代码密度。这项技术通过在相同的指令流里面混合32位和16位的指令流来提高代码密度。要达到这个目标，开发人员需要在处理器里面支持对非对齐地址的访问。

- **增强型的DSP指令：**在标准的ARM指令集上增加的这些指令可以实现灵活快速的16 × 16乘法和饱和算术指令，这有助于面向DSP的函数可以方便地移植到ARM上。一个单ARM处理器可以不需要单独的DSP就能满足VoIP的需求。举例来说，ARM处理器可以使用SMLAxy指令来对32位寄存器的高16位和低16位相乘。处理器可以把寄存器r1的高16位和寄存器r2的低16位相乘，然后把结果保存到寄存器r3中。

图1表示了饱和运算如何影响ADD指令的计算结果。饱和运算非常适合于数字信号处理，如果使用非饱和运算可能会发生整形溢出回绕而导致计算结果为负值。饱和运算指令QADD可以返回整形的最大值而不会发生回绕。

Nonsaturated (ISA v4T)	Saturated (ISA v5TE)
PRECONDITION	**PRECONDITION**
r0=0x00000000	r0=0x00000000
r1=0x70000000	r1=0x70000000
r2=0x7fffffff	r2=0x7fffffff
ADDS r0,r1,r2	QADD r0,r1,r2
POSTCONDITION	**POSTCONDITION**
result is **negative**	result is **positive**
r0=**0xefffffff**	r0=**0x7fffffff**

▲ 图1 饱和和非饱和加法。饱和运算非常适合于数字信号处理，如果使用非饱和运算可能会发生整形溢出回绕而导致计算结果为负值。

数据层的并行化

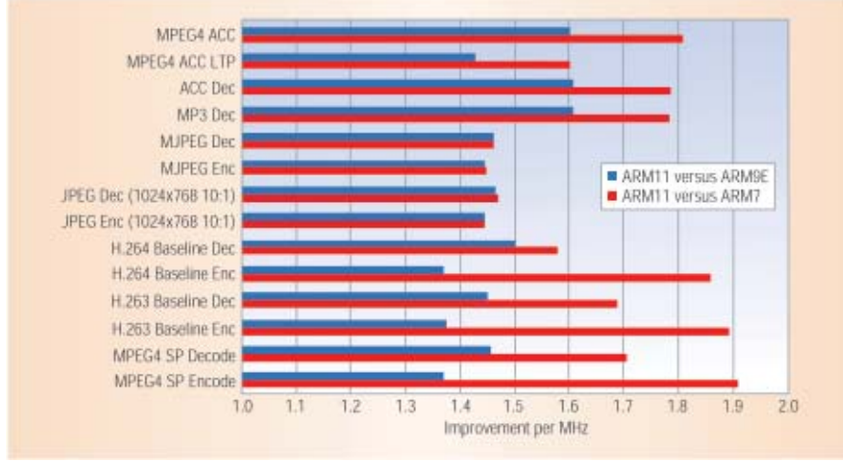
在v5TE架构下的DSP增强指令取得成功以后，ARM在2001年发布了ARMv6架构。除了增强数据层面和线程层面的并行化以外，在算术运算，异常处理和对齐（endian-ness）处理方面也有改进。

影响ARMv6架构设计的一个重要因素是针对视频处理和2D/3D图像处理提供了更多的DSP指令。ARMv6成功地增加了这些新的功能，并保持了原有的低功耗的特点。ARM采用了单指令多数据（SIMD）架构来实现这些增强。

SIMD在不损害代码密度和功耗的前提下，可以成功的提供数据层面的并行化。SIMD实现可以用相对很少的指令来完成复杂的计算，并且只需要很少的内存访问。

ARM SIMD在计算效率和低功耗之间取得了很好的平衡，它可以把标准的32位数据路径分割成4个8位的，或者两个16位的数据路径。这一点和很多其他的实现不同，那些实现往往需要额外的特殊数据通路来完成SIMD操作。

图2表明了ARMv6架构下在ARM11处理器上使用SIMD，不同编解码器的性能改进



▲ 图2 SIMD指令和非SIMD指令的功耗。ARM实现的轻量级SIMD指令减少了使用的门数目，显著降低了硅片面积、功耗和系统的复杂性。

ARM 的 SIMD 实现使用较少的逻辑门数，极大地减小了硅片面积、降低功耗和复杂性。并且所有的SIMD指令都可以条件执行。

为了改进对视频压缩算法（例如MPEG 和H.263）的处理能力，ARM 引入了差值绝对值累加和运算（SAD）。如运动估计运算，通过计算

$$SAD = \sum | R(I,j) \quad C(I,j) |$$

来比较两个像素块，小SAD 值表明这两个像素块可能是相似的像素块。因为运动估计要对不同的相对地址进行多次SAD运算，视频压缩系统需要功耗效率高的快速 SAD 运算。

USAD8 指令和 USADA8 指令可以对 8-bit 数值进行差值绝对值累加和运算。这个特点对运动视频压缩和运动估计算法非常有用。

Table 1. Exception handling instructions in the ARMv6 architecture.

Instruction	Description	Action
CPS	Change processor state	CPS<effect> <iflags>,{,#mode} CPS #<mode> CPSID <flags> CPSIE <flags>
RFE	Return from exception	RFE<addressing_mode> Rn!
SRS	Save return state	SRS<addressing_mode>,#<mode>{!}

ARMv4T ISA	ARMv6 ISA
; Copy CPSR	; Change processor state and modify
MRS r3, CPSR	; select bits
; Mask mode and FIQ interrupt	CPSIE f, #SVC
IC r3, r3, #MASK|FIQ	
; Set Abort mode and enable FIQ	
ORR r3, r3, #SVC|nFIQ	
; Update the CPSR	
MSR CPSR_c, r3	

▲ 图 3 完成同样的改变处理器状态的功能，ARMv6 架构的指令和ARMv4T 架构下的指令的比较。

线程层面的并行化

我们可以把线程看成具有私有 PC 指针和寄存器组、但有着相同内存空间的进程。ARM需要改进异常处理来为越来越复杂的多线程和多处理器结构提供支持。这些需求增加了中断处理、调度器和上下文切换在数据一致性方面的复杂性。

为了减少对时间非常敏感的上下文切换时间，ARM 对异常处理指令也进行了优化：在指令集中增加了三条指令，如表一所示。

如图3所示，程序员可以使用改变寄存器状态指令（CPS: change processor state）来修改当前的CPSR 寄存器直接进入Supervisor 模式并禁止快速中断响应.同样的功能在ARMv4T架构下需要4条指令才能完成，而在ARMv6架构下只需要两条就可以了。

程序员可以使用保存返回状态指令（SRS: save return state）在特定的模式下改变SPSR。在ARMv4T 机构的特定模式下修改 SPSR 比在 ARMv6 机构下需要更多的指令。这个新指令在上下文切换和异常处理代码返回时非常有用。

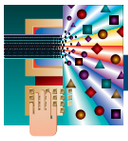
多处理器上的原子操作指令

在早期的ARM处理器上是用swap指令实现旗语（semaphore）的，它可以挂起外部总线直到指令执行完成。显然，这对于线程级别的并行化是无法接受的，一个处理器会把整个的总线挂起直到指令执行结束，在这期间其他处理器的执行也停止了。ARMv6新增加了两条指令：排他数据加载（load-exclusive）指令LDREX和排他数据保存（store-exclusive）指令STREX，他们利用了内存中的监控器（exclusive monitor）来达成目标。

ARM11处理器是对ARMv6的最早的硬件实现。它内含8级流水线，针对加载/存储和乘/累加有独立的并行流水线。通过并行的加载/存储单元，ARM1136J-S处理器可以不用等待慢速内存而继续执行，这直接提高了处理器的性能。

除此之外，ARM1136J-S 处理器还使用物理地址标记的缓存来支持线程级别的并行机制，这与之前的 ARM 处理器使用虚地址标记缓存截然不同，可以极大的改善大型操作系统下的上下文切换性能。

使用虚地址标记缓存，因为缓存中包含了旧的虚地址到实地址的转换关系，所以每次上下文切换的时候都要清理（flush）缓存。在ARM11 中，内存管理单元（MMU）位于一级缓存和处理器内核之间，这样就避免了使用虚地址标记缓存带来的每次上下文切换时要清理缓存的问



题，减少了对外部内存的访问次数，降低了整体的功耗。实地址标记缓存对系统的整体性能可以提升 20% 左右。

指令级的并行机制

对一个指令序列，处理器可以同时执行多条指令。这种形式的并行化可以在不改变软件编程模型的前提下可以显著提高处理器的整体性能。

显然，指令级的并行更加强调利用编译器来调度指令序列以支持超标量内核。尽管好的编译器可能简化硬件设计的复杂性，但是由于硬件需要侦测并行代码和提高运行频率来达到高性能，复杂性和成本不可避免地会增长。

ARM 在网络计算机领域已经保持了数年的领先地位。这个技术领域一直变化很快，大型机上的技术不断的被转移到小型处理器中。例如，大型机十年到二十年前的技术现在被广泛的应用到今天的台式机中。同样，五年前针对台式机的技术也开始在消费类电子产品和网络产品中。例如，对称多处理器技术（SMP）在现今的台式机和嵌入式计算机中都能找到。

性能和功耗

在过去的几年里面，消费者不断提出需要在他们的嵌入式设备里面拥有类似台式电脑的功能，嵌入式处理器市场因而不断采用桌面计算机中的成熟技术。然而，要求用低功耗得到高性能的需求，使得嵌入式处理器的发展趋向于使用多处理器和硬件加速器来降低系统的整体功耗。当今，对高性能的通用处理器的需求使得在嵌入式系统和桌面系统中都开始采用 SMP 作为应用处理器。

2004 年，嵌入式和桌面市场都遇到了用提高主频的方法来提高系统性能的障碍。对此，开发人员开始转向SMP技术来规避一下一些难题：

- **高主频意味着高功耗。**提高处理器的主频对处理器的功耗的影响是成平方关系的。处理器主频的翻倍不仅意味着电路反转的动态功耗翻倍，还意味着需要更高的工作电压，这些意味着系统的整体功耗按照系统频率的平方关系增长。更高的主频还增加了设计的复杂性，极大地增加了处理器需要的逻辑门电路的数量。
- **硬件侦测指令并行的技术更为复杂和昂贵。**使用硬件来生成并行指令极大地增加了硅片面积和设计的复杂性，并进而增加了功耗。
- **对多个独立处理器的编程是不可移植的和低效率的。**开发人员需要面对不同架构的多个处理器，软件复杂性的增长降低了软件的可移植性。

在2004年中期，PC厂商和芯片制造商一系列的通告预示了在桌面计算机市场主频竞赛的结束和在服务器领域多内核 SMP 处理器应用的开始，Intel 的超线程 Pentium 处理器充分证明了这一点。与之同步的是，ARM 发布了 ARM11 MPCore 多内核处理器产品，为解决性能可扩展性提供了一个重量级的解决方案。

伴随 ARM11 MPCore 同时发布的还有对 ARMv6 架构的一系列增强，它们可以对先进的 SMP 操作系统提供更广泛的支持。ARM 把这些增强特性命名为 ARMv6K 或者 AOS(Advanced OS Support)。这些增强特性存在于所有的基于 ARMv6 架构的应用处理器中，为嵌入式软件提供了坚实的基础。

ARM11多内核处理器解决了SMP系统中的两个主要的设计难题：

- 在集成了新的 ARM 通用中断控制器之后的处理器内核之间的通信问题
- 通过引入监听控制单元（SCU，一个智能内存通信系统）后解决了缓存一致性的问题

这些逻辑模块提供了一个高效率、统一的SMP处理器，生产厂商可以获得很高的成本效率。

使用 ARM 多内核处理器的准备

在通用计算的环境下要充分利用多处理器内核硬件平台的优点，ARM 需要提供一个缓存一致的、对称的软件平台，它需要具有丰富的指令集。在开发过程中，先进的 ARMv6 架构下的一些增强特性可以提供非常大的性能提升。

增强型的原子指令

研究人员可以使用 ARMv6 的排他加载 / 存储机制实现基于交换的旗语和基于比较 — 交换的旗语来控制对关键数据的访问。在传统的SMP服务器计算领域，在软件上使用上锁 – 释放同步机制，对优化 SMP代码已经有了很多的投入。这些工作主要集中在x86架构下，开发人员可以使用它的原子指令来比较和交换数据。

许多开发人员喜欢在上锁–释放子例程中使用 Intel 的 cmpxchg8b 指令，它可以以原子方式交换和比较 8 字节数据。通常，4字节用于有效数据，4字节用于区分不同有效数据，这个方式主要用来解决有效数据相同的问题。这种问题通常也被称为 A–B–A 问题。

ARM 的互斥机制通过数据地址而不是数据本身提供原子操作能力，这样例程就不用解决 A–B–A 问题而直接以原子方式来交换数据。然而，要利用这一点，需要重写现在的双字互斥代码。ARM 提供的可以进行排他加载 / 保存指令可以使用任何长度的有效数据 – 包括8字节 – 这样就保证了现存的多线程代码可以直接的进行移植。

对局部数据的访问的改进

当OS在SMP平台上遇到大量的多线程应用时，它必须考虑在线程状态的交互和当前正在执行的线程上的性能开销。举例来说，这包括需要知道一个线程正在哪个CPU上执行，访问线程相关的内核数据结构，和使能线程对局部数据的访问。AOS 增加了寄存器来帮助解决这些 SMP 性能方面的问题。

CPU 编号

使用标准的 ARM 协处理器接口，处理器上的软件可以使用一个简单的非内存访问来识别它在哪个处理器上运行。开发人员可以使用这个数据作为访问内核数据结构的索引。

上下文寄存器

SMP 操作系统提供对线程相关数据的访问是可以解决两个关键问题。ARMv6K 架构扩展定义了三个额外的系统协处理器寄存器，OS 可以将其用于任何合适的目的。每个寄存器有不同的访问级别：

- 用户模式/特权模式下的读写访问
- 用户模式下只读，特权模式下可读写
- 仅特权模式下可读写

对这些寄存器的使用随操作系统而不同。对 Linux 内核和 GNU 工具链，ARM 应用二进制接口使用这些寄存器来保存线程的局部数据。线程可以使用TLS不占用任何通用寄存器来快速的访问线程相关的内存。

为了在 C 和 C++ 之中支持 TLS，定义了一个新的关键字 thread，它可以在定义和声明变量的时候使用。尽管还不是官方的扩展，许多的编译器都已经开始支持这个关键字了。通过这种方式定义或者声明的变量可以自动分配为线程局部访问的数据：

在随Linux2.6内核发布的新的native POSIX线程库中，支持TLS是一个关键的特性。与老的Linux pthread 库相比，新的POSIX线程库可以极大的改进性能。

考虑了功耗问题的旋转锁

SMP 系统中的另一个问题是多处理器需要访问共享数据时的线程之间的同步开销。在许多 SMP 同步机制的最低的抽象层中，软件旋转锁使用内存中的一个变量作为锁。如果指定的内存中是一个已经预定义好的值，OS就认为共享资源被锁住了；如果情况相反，OS 就会认为共享资源是可用的。在软件访问共享资源之前，它必须先通过一个原子操作得到锁。当软件结束对共享资源的访问后，它必须释放锁。

在SMP OS中，当一个处理器占有锁的时候，另一个处理器必须等待。旋转锁的得名就是来源于处理器在等待期间必须用一个短循环不断地试图获得锁。后来的改进包括使用一个后退循环来减少总线竞争，在这个后退循环中处理器不再试图访问锁。对任何一种情况，这些没有任何产出的指令指令周期都消耗了能量。

AOS 扩展包含了一对新的指令，它们可以在处理器等待锁的时候进入睡眠，这样就可以减少能量的消耗。ARM11 多处理器结构实现了这些指令，它可以在锁内释放后对等待的处理器提供一个通知，根本就不需要一个回退循环。这不仅可以减少能量损耗，还提供了一种高效率的旋转锁实现机制。

弱排序的内存一致性

ARMv6 架构对不同的可定义内存区定义了各种内存一致性模型。在 ARM11 多处理器中，旋转锁的实现代码使用具有缓存一致性的内存来保存锁的值。作为一个多核处理器，ARM11 MPCore 是第一个对程序员公开弱排序内存的 ARM 处理器。它使用三条指令来控制弱排序内存的作用：

- wmb(). 这个Linux宏创建了一个写内存屏障，多处理器可以在一个对这个屏障指令附近的内存写序列中放置一个标记。旋转锁可以在解锁之前执行这个指令来确保对有效数据的任何写操作都能在释放旋转锁之前完成，因此也肯定在其他处理器获得锁之前完成。为了保证更高的性能，这个屏障不需要通过清空数据来暂停处理器，而是通知 load-store 单元并且能在大多数情况下让指令的执行继续进行。

- rmb(). 在 Linux 内核中，这个宏通过放置一个读内存屏障来防止任何读操作在取得锁之前可以读到有效数据。尽管在在ARMv6架构下这完全合理，这种级别的弱排序内存很难保证软件的正确性。因此，ARM11 多处理器仅实现了非投机性的预读操作。当存在着不需要进行读操作的可能性时，例如旋转锁中的情况，在teqeq指令和任何的有效数据读之间存在着一个跳转指令，预读操作不会发生。所以，对 ARM11 MPCore 多内核处理器，这个宏可以被定义为空。

- DSB (Drain Store Buffer). ARM架构包括了一个仅对处理器可见的内存。当运行单内核的软件时，处理器允许扫描该缓存中的数据。然而，在多处理器系统中，这个缓存对其他内核的读操作就不可见了。DSB 可以把这块缓存的内容清理到 L1 缓存中，在 ARM11 多处理器中，L1 Cache 对多处理器是一致的。清理操作之需要在其他处理读数据之前写到 L1 Cache 就可以了。在旋转锁的解锁

```
static inline void _raw_spin_lock(spinlock_t *lock)
{
        unsigned long tmp;

        _asm__ __volatile__(
         1: ldrex %0, [%1]                          ; exclusive read lock
            teq %0, #0                              ; check if free
            wfene                                   ; if not, wait (saves power)
            strexeq %0, %2, [%1]                    ; attempt to store to the lock
            teqeq %0, #0                            ; Were we successful ?
            bne 1b                                  ; no, try again
        :     "=&r" (tmp)
        :     "r" (&lock->lock), "r" (1), "r" (0)
        :     "cc", "memory"
   );

        rmb();          // Read memory barrier stops speculative reading of payload
}                       // This is NOP on MPCore since dependent reads are sync'ed

static inline void _raw_spin_unlock(spinlock_t *lock)
{
        wmb();          // data write memory barrier, ensure payload write visible
                        // Ensures data ordering, but does not necessarily wait
        _asm__ __volatile__(
          str %1, [%0]                              ; Release spinlock
          mcr p15, 0, %1, c7, c10, 4                ; DrainStoreBuffer (DSB)
          sev                                       ; Signal to any CPU waiting
        : "r" (&lock->lock), "r" (0)
        : "cc", "memory");
}
```

▲ 图 4 为 ARM Linux 2.6 内核中旋转锁的加锁和解锁实现的范例代码。

代码中，处理器在SEV指令之前立即执行DSB就可以保证任何处理器在唤醒时可以读到正确的值。

ARM11 MPCORE 多内核处理器

指令集的适配性并不是唯一的影响多处理器提供SMP带来的可扩展性前景的因素。如果实现不好，SMP设计的两个方面可以极大地限制系统的最大性能并增加能量的消耗。

图 4 考虑了功耗的旋转锁。范例代码为ARM Linux 2.6内核中旋转锁加锁和解锁的代码。

- Cache 一致性。开发人员一般提供需要Cache一致性的单映像SMP OS，所以它可以通过把数据放在Cache中维持稳定的性能。在ARM11多内核处理器中，每个CPU有自己的指令和L1数据Cache。现有的维持一致性的方案通常通过增加信号扩展系统总线来控制和检测其他CPU的cache。在嵌入式系统中，系统总线的时钟通常慢于CPU。这样，除了在处理器和cache之间造成瓶颈，这种办法极大的增加了总线上的流量和总线的功耗。ARM11 MPCore通过设计了一个多处理器之间的智能SCU解决了这些问题。运行在CPU频率上，这种方法提供了快速的数据路径，使数据在各个CPU的cache之间快速移动成为可能。

- 处理器之间的通信。SMP OS需要在CPU之间进行通信，最好的实现方式是不通过存储器访问来完成。系统必须通过旋转锁来同步处理器之间的通信和对共享资源的访问。其他的SMP OS CPU之间的通信可以不用访问内存来完成。异步运行的系统之间必须经常进行同步。其中的一个方法是通过中断系统来触发另一个处理器上的活动。这种软件引发的处理器之间的中断一般使用一个特殊的中断系统，它被设计成通过I/O外设来触发中断而不是直接由CPU来触发。

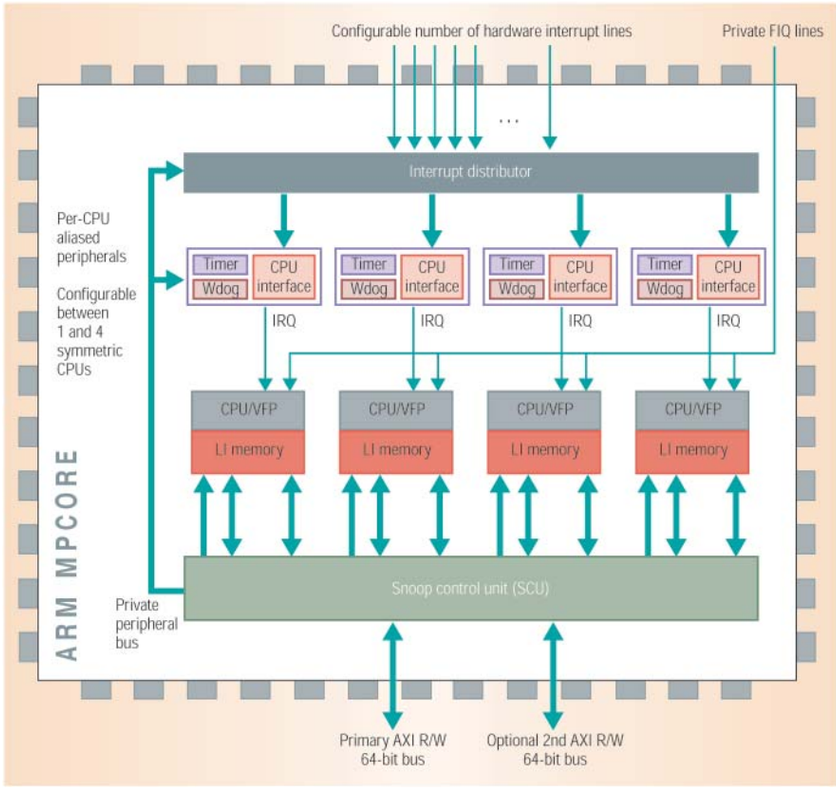
中断子系统

下面我们举一个在SMP中使用IPI的例子，假设一个多线程应用程序，在某一个处理器上运行的线程会改变处理器的状态，并且这种改变对该应用程序运行在其他处理器上的线程不是硬件一致的。如果应用程序分配新的虚地址空间这种情况就会发生。为了维护一致性，OS必须把这些对内存映射的修改同步到其他处理器中。在这个例子里面，OS一般把新的内存映射配置到处理器中，然后用低竞争的私有外设总线写GIC中断控制寄存器，GIC会触发其他处理器上的中断。其他的处理器会根据中断ID来判定它们需要更新各自的内存映射表。

GIC可以通过中断分发单元，使用各种软件定义的模式来把中断导向特定的处理器。除了可以对应用程序进行动态负载均衡以外，SMP OS通常也对中断处理进行动态负载均衡。OS可以使用挂在私有外设总线上的每个处理器单独的别名控制寄存器来快速的改变任何一个特定中断的目标寄存器。

另外一种管理中断分发的方案是把中断发送到一个定义好的处理器组。MPCore会认为当前负载最轻的处理器最适合处理中断，并把中断发往当前负载最轻的处理器。这种灵活的处理方式使GIC技术在很多ARM处理器上都能够得到广泛的应用。该标准，也简化了软件如何和中断控制器进行互操作。

监听控制单元（SCU: Snoop Control Unit）



▲ 图 5 展示了ARM11 MPCore集成在内核内部的ARM GIC如何使中断系统的访问更有效的完成。ARM设计了GIC来优化SMP OS中关键部分IPI（interprocessor interrupts）的成本。

MPCore的SCU单元是一个主要用于控制多处理器之间数据cache一致性的智能控制单元。每次内存发生更新的时候，需要监督和操控每个处理器cache的内容，为了限制这些操作对功耗和性能的影响，SCU保留了每个缓存行（cache line）的物理地址标签的副本。有了本地保存的这些数据的副本，SCU就可以降低对有着共同缓存行的处理器进行cache操控的机率。

处理器通过优化的MESI（modified, exclusive, shared, invalid）协议来维护cache的一致性。根据MESI，一些常见的操作，如A = A + 1，当对共享数据进行操作时会引发一系列的许多状态转换。

为了改进性能和降低维护cache一致性上的功耗开销，SCU可以智能的监督处理器对缓存行的操作。如果一个处理器修改了一个缓存行，另一个处理器对该缓存行先读后写，SCU就会假定这个地址今后也会经历同样的操作。如果同样的操作再一次发生，SCU会自动的把缓存行的状态置为无效，而不是消耗能量先把它置为共享状态。这项优化在不引发外部内存操作的情况下，让处理器把缓存行直接传递到其他的处理器上。

这种把共享数据直接在多个处理器之间转移的能力为程序员提供了一个可以用来优化软件的先进的特性。当定义处理器之间会共享的数据结构时，程序员需要确保合适的数据结构的对齐和压缩，来保证缓存行转移的发生。如果程序员使用队列来在多个处理器之间分发工作，他们需要确保队列有一个合适的长度和宽度，这样当一个处理器选中一个工作项时，它会把这个工作项通过这种cache到cache的转移机制继续传递。为了帮助完成这个级别的优化，MPCore包含了硬件结构来支持包括传统L1 Cache和SCU的很多操作。

ARMv6K结构可以被认为是一个充分考虑了多处理器结构的重要的指令集。在它奠定的低功耗设计的基础上，ARM11 MPCore的架构和实现能够给高性能设计带来低功耗的特点。这些新设计将给人们如何使用技术带来深远的影响。目前每年有超过15亿的ARM处理器被售出，对基于ARM的开发人员来说他们的软件代码将存在着不可估量的市场机会。

References

1. D. Seal, ARM Architecture Reference Manual, 2nd ed., Addison-Wesley Professional, 2000.
2. A. Sloss et al., ARM System Developer's Guide, Mor-gan Kauffman, 2004.

John Goodacre is a program manager at ARM with responsibility for multiprocessing. His interests include all aspects of both hardware and software in embedded multiprocessor designs. Goodacre received a BSc in computer science from the University of York. Contact him at john.goodacre@arm.com.

Andrew N. Sloss is a principle engineer at ARM. His research interests include exception handling methods, embedded systems, and operating system architecture. Sloss received a BSc in computer science from the University of Hertfordshire. He is also is a Chartered Engineer and a Fellow of the British Computer Society. Contact him at andrew.sloss@arm.com.