

Overview

- operating system issues
 - background / review
 - multicore issues and goals
 - scheduling and work distribution
 - synchronization support (adaptive locking, schedctl)
 - interrupt handling; memory management
- intermission
- virtualization
 - overview, motivations & objectives
 - mechanisms: the hypervisor, control domain
 - memory and device management
 - hardware support on the T2; wallman configuration
- cryptographic accelerators
- refs: [Hughes&Hughes, Ch4 – 6]; OpenSPARC-Slide-Cast (Part 09)

Developing and Tuning Applications On Ultrasparc T1 Chip Multithreading Systems, Denis Sheahan, Sun BluePrints OnLine, October 2007.

T2 Solaris Changes, Denis Sheahan, Sun; Xen and the Art of Virtualization, Barham et al, SOSP'03

Background: OS Support for Threads

- idea of threads: 'lightweight' processes that principally share an address space
- thread memory layout; motivation for threads (4300 Lect 11, p3)
- thread-related services:
 - creation and termination (exit, cancel a peer thread)
 - scheduling: transitions from runnable, running, sleeping (or parked) and stopped
 - ◆ system calls: yield (self); 'park' (other threads)
 - ◆ may be used to support efficient synchronization (e.g. adaptive spinlocks)
 - OS must also deliver signals on a thread level
 - thread state includes: id, status, attributes (e.g. scheduling policy & priority), register contents, stack [Hughes&Hughes, Fig 6-2]
 - ◆ OS must keep track of this; save / restore (except stack) upon going to /from running status
 - for multicore, kernel threads (parallelism) are of main interest (vs user threads (programmability; hiding delays); see 4300 Lect 11, p13-15)

Operating Systems: Background

- software developer's interaction with the OS [Hughes&Hughes, Fig 4-1]
- relevant core services:
 - process (thread) management: processes and scheduling; process state, context switching (6300 Lect O1 p6–10)
 - memory management: paging & executable memory image (6300 Lect O2, p4,15)
 - device management, via memory-mapped I/O (6300 Lect O4, p6)
- OS services are provided by system calls (e.g. I/O) (6300 Lect O4 p8,10)
 - the trap instruction (or an exception, or an interrupt) cases a switch from user mode to kernel mode; (6300 Lect P9, p3,8)
corresponding retry or done instruction to return to user mode
 - user mode: access memory (instructions and data) within process' address space; can use non-privileged (normal) instructions and registers
 - kernel mode: access also kernel's address space (including PCB and page tables) as well; can also use privileged instructions and registers (including those to manipulate caches, TLBs, etc)
 - may need to save user-level state (e.g. non-privileged registers) to service the trap (depending on how complex this is)

Operating System Issues for Multicore

- OS's role: safely and fairly manage system resources
- for multicore/multithread:
 - have many more resources, with a complex hierarchy: many virtual CPUs (vCPUs), with shared (e.g. L2\$ and semi-shared (e.g. L1\$, (e.g. TLBs) state
 - require a high degree of threading; OS has a greater role in performance!
 - goal is aggregate, rather than single thread, performance; issues include:
 - ◆ threads on same core: if one is stalled on L2\$ miss, other threads can still run (constructive sharing)
 - ◆ if one thread is thrashing a cache or TLB on a core, other threads on core can be adversely effected (destructive sharing)
- all OS mechanisms (e.g. access to kernel data structures) must be highly scalable (e.g. use scalable locks or if possible lockless methods)

Scheduling for Multicore

- OS will organize vCPUs into a hierarchy of processor groups: a group of vCPUs sharing cores (IU / FPU pipelines, L1\$, MMU) or sockets (L2\$) (Sheahan T2/Sol p10)
- tries balance load across the chips, then the cores, then the pipelines etc
 - e.g. run `mpstat` for a 24 thread job on `wallaman`
- try to maintain **affinity**: restart thread on the last vCPU it ran on
- otherwise, must **migrate** (to another vCPU): causes thread to pull data into cold caches / TLBs (possibly affecting other threads on core)
- (on the T2) if a vCPU has no work: enters the idle loop in the kernel, kernel then parks the vCPU in the core (no longer candidate for H/W scheduling)
 - a thread with **real-time class priority**: may force parking of other vCPUs in the core (to give priority)
- OS must also provide support for binding processes/threads to cores (e.g. Solaris `processor_bind()`, Linux `get/set_cpu_affinity()`)
- for I/O bound applications, may need a large thread pool size to hide I/O latency (starting point: # vCPUs); OS must be able to efficiently support this scenario

Interrupt Handling

- interrupt handling: a single vCPU may give a slower response (compared with a non-threaded core)
 - solution: 'fanout' interrupt processing using either hardware or software
- via software: the vCPU handling the interrupt (for say a network device) simply deposits packets into an appropriate queue
 - a kernel thread can be scheduled on any other vCPU
 - e.g. Solaris GigaSwift Ethernet driver: has 4 queues
- interrupt handling tends to be highly disruptive (switch to kernel context: will 'pollute' TLBs and caches)
- solutions:
 - OS can consider handling all on one core; avoid scheduling compute-bound threads to vCPUs handling interrupts
 - also manual support to disable interrupts (e.g. Solaris: `psradm -i cpuid`; `psradm` also used for taking vCPUs on/off line)

Synchronization Support

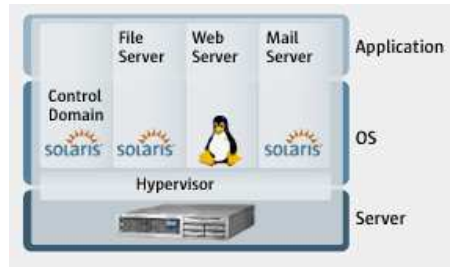
- adaptive locks: user-level lock routines (e.g. `pthread_mutex_lock()`) will spin for a time period before asking kernel to handle lock acquisition
 - kernel may similarly spin before deciding whether to transition thread to sleeping
- CMT concept: threads are plentiful but less powerful; thus a thread is more likely to be running
 - spin times can be tuned accordingly (esp. inside kernel)
- also undesirable to deschedule running thread holding any resource (e.g. lock) that other threads may be waiting on
 - OS can provide a system call to advise avoiding pre-empting the thread (e.g. Solaris `schedt1` calls)
 - critical sections can be bracketed by system calls to advise / un-advise not to pre-empt

Memory Management for Multicore

- OS does lots of memory copying; needs fast memory movement routines
- page coloring: L2\$ lines are indexed by the physical address (determined by OS)
 - virtual to physical address mappings can affect conflicts in the L2\$
 - very difficult to predict this with a large number of threads!
 - solution: mitigate by hardware support instead
 - ◆ have a highly associative (e.g. 16-way) L2\$ and (on T2) hash (xor) low order indexing bits with high-order (tag) bits to randomize L2\$ indexing
- optimizing page size selection is also important (tradeoff – larger: fewer TLB misses, but more fragmentation). E.g. on Solaris/T2:
 - thread stack: 8KB; text, main stack, (small) heap: 64 KB; 4MB large heap
 - can be viewed by `pmap -s pid`
- when running multiple copies of the same binary, (commonly used) variables in stack and data areas have same address
 - causes conflicts in L1\$ and L2\$
 - can be mitigated by stack biasing: randomly offsetting base of main stack in different processes

Virtualization: Overview

- can put several logical servers (operating systems) on a single physical system
- allocate resources (CPUs, memory) to each; can vary dynamically
- typically share disks and network interfaces
 - each virtual machine will be given a different 'partition'
- typical Xen configuration (courtesy Muhammad Atif, ANU)
- multicore: can easily allocate a logical server on each vCPU
 - T2 via logical domains can support up to 64!; H/W support reduces overhead
- each logical server can be for a different application / client
 - high degree of security / isolation possible
- many other flavors of virtualization available (e.g. Xen, VMware, KVM)



Virtualization: Mechanisms

- achieved by introducing a third privilege level in the CPU, in which a hypervisor manages machine resources (CPUs, memory, devices) through manipulating vital processor state (e.g. TLB and IO-TLB contents) (OS SlideCast Ch9 p4)
 - more like a 'nucleus' than an operating system; provides minimal services to guest OS via hyper-calls (OS SlideCast Ch9 p10,11)
 - also called a virtual machine monitor (VMM); guest OSs called virtual machines (VMs)
 - can re-bind VMs to physical resources at any (?) time
- main approaches are full virtualization (n.b. all modern processors support this!)
 - can run any guest OS unmodified! (do not need to trust it!)
 - performance penalties (e.g. 'trap' to VMM when VM tries to update TLB entry) and para-virtualization (Xen, Logical Domains)
 - guest OS is modified and is aware of hypervisor
 - ✓ some performance advantages (e.g. ask VMM to fill in page table in one go)
 - ✗ large maintenance burden if not supported by main developers (e.g. XenLinux)
- usually, there is a special VM (control domain or domain 0) which is used to configure and control other guests (OS SlideCast Ch9 p6-7, Xen)

Virtualization: Motivations and Objectives

- enables *consolidation* of services (single machine can host multiple logical services)
 - e.g. data center hosting many web servers (fewer resources needed provided different peak times), export part of machine to different user groups
 - also each client can have own custom OS (can be self-configured and managed)
- high degree of encapsulation enables *migration* to another (compatible) physical machine
 - improved reliability, less downtime, optimize total system utilization/power according to current demand
- goals:
 1. provide isolation: both data and performance
 2. support a variety of (guest) operating systems
 3. provide a low performance overhead

Virtualization: Memory Management

- without virtualization, the OS performs address translation as the mapping:
$$f(VA, c) = PA$$
where c is the current context id (held in a particular privileged register)
 - each current process (and the kernel) will have a different value of c
- under virtualization, the VM performs $f(VA, c) = RA$, where RA is called the 'real address'
 - which the VMM then translates via $g(RA, p) = PA$, where p is the partition id (OS SlideCast Ch9 p14,17)
 - each VM will have a different value of p
- TLB entries must contain the PA; page table entries must contain RAs under full virtualization
 - loses potential for VM to improve performance (e.g. for page coloring)
- note: it is possible for multiple VMs to share a single real CPU (time-slicing)
 - issue: do you provide a real or virtualized notion of time?

Device Management under Virtualization

- under Xen, VMs have modified device drivers which copy memory to shared memory buffers
 - Domain 0 drains these buffers, the real device drivers sending data to the real device (Xen, Logical Domains OS SlideCast Ch9 p30)
 - considerable overhead: VMM must swap TLB entries for buffers from VM to Domain 0 (page flipping)
- it is also possible give guest VM direct control of a PCI (disk or network) device (OS SlideCast Ch9 p20)
 - VM directly accesses real device (exclusively!) with a real driver (para-virtualization)
 - VMM virtualizes access to the IO-MMU, causing VMM intervention whenever VM attempts to access device (full-virtualization)
 - the IO-MMU is the Memory Management Unit associated with the PCI bus
 - translates PAs in the PCI address range into accesses to the appropriate device (memory-mapped I/O)
- in all cases, when (virtualized) I/O transfer completes, VMM directs interrupts to the relevant VM

Virtualized Address Translation on the T2

- tag part of each TLB entry contains:

context id	partition id	r	VA
63	48	47	45 44 41 0

data part includes the corresponding PA (for top of page frame in physical memory)

- Memory Management Unit supports both VA → PA and VA → RA
 - if the r (real) bit is 1, the context id is ignored
- direct access to MMU control registers (including TLB entries) is allowed only in hyper-privileged mode
- in the guest OS (VM), only VA → RA mappings are stored in the page tables
- Q: how can different processes (and VMs) share the same core?

(hint: the caches' tags are PAs, not VAs)

Hardware Support for Virtualization on the T2

- each vCPU supports six trap levels; the top 4 are hyper-privileged mode only
- privileged registers (accessible to guest OS kernel) include:
 - process state register: contains privileged, interrupt enabled bits etc
 - trap state register stack, indexed by a trap level register, containing process state register, PC, trap type etc for the vCPU at previous traps
 - context register (13-bit), indicates the (process-level) context to interpret a VA
 - (priv.) trap base register: base of trap table in (guest) kernel memory
- hyper-privileged registers (directly accessible only to hypervisor) include:
 - hyper. process state register: contains a hyper-privileged bit
 - various vCPU status registers (chip-wide; can be used to take on/off line)
 - 3-bit partition identifier register indicates which guest OS is currently running
 - interrupt pending register, (hyper-priv.) trap base register, perf. counter registers
- (from user mode) traps to privileged mode include:
 - attempt priv. operation, arithmetic exceptions, register window spill, software interrupt, performance counter register overflow

but (most) traps go to hyper-privileged mode:

 - reset, hardware interrupt, attempt hyper-priv. operation, memory access related exceptions (I/D-TLB miss, invalid addressing, data protection error)

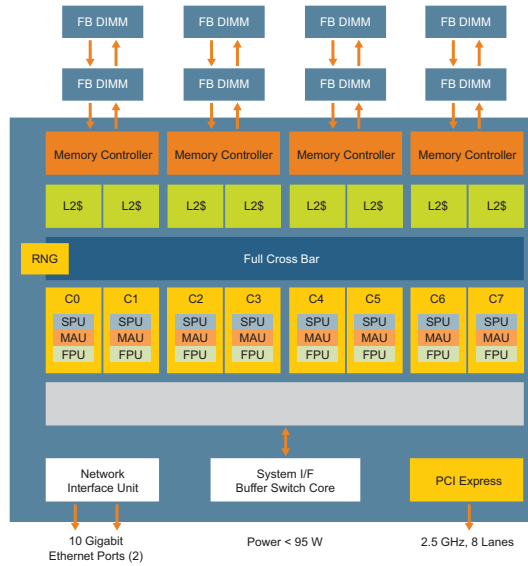
wallaman Configuration

- control domain is mavericks (subnet 150.203.163): T5120 with 32GB RAM, 2 XAU network interfaces, 2 × 146 GB disks
- wallaman is exported from mavericks as logical domain ldom1. Currently has:
 - 8GB RAM (RA 0x8000000 → PA 0x48000000)
 - 56 vCPUs, virtual console on port 5000
 - 30GB virtual disk (mavericks: /export/zpool/ldom1/vdisk1_30gb.img)
 - network interface (vnet1) has direct access to interface mavericks:e1000g1, on subnet 150.203.24
 - why isn't it on subnet 150.203.163?



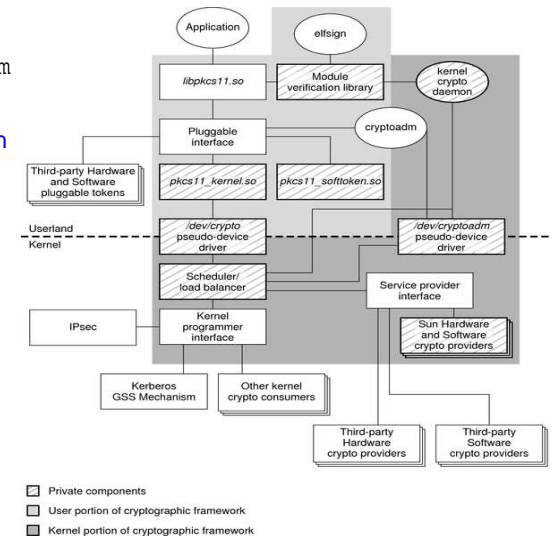
Hardware Accelerators on Multicore: T2's Crypto Units

- motivation: 'zero-cost security' (negligible performance impact)
- why on-chip accelerators: avoid extra I/O, reduce latency, less power
- T2 system design: have Modular Address and Streaming Processor Unit with each core (courtesy *Transparent Multi-core Cryptographic Support on Niagara CMT Processors*, Hughes et al)
- per-chip Random Number Generator



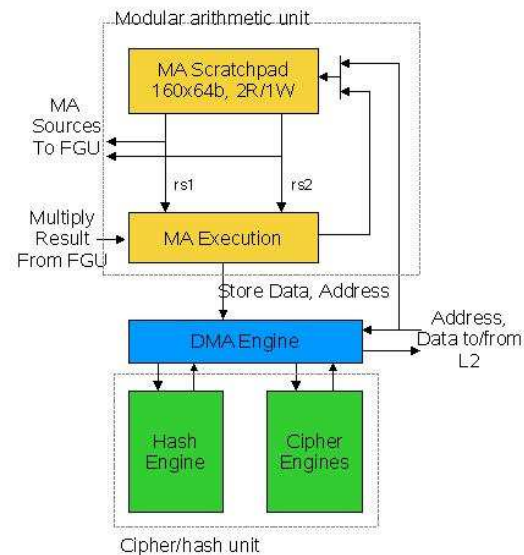
T2's Crypto Units: Solaris Cryptographic Framework

- diagram courtesy of Sun article
- can disable `/dev/crypto` (NCP & N2CP) via `cryptoadm` command; `softtoken lib.` provides a s/w implementation
- asymmetric key request can be scheduled (round robin) to run on different cores
- considerable overhead in accessing SPU / MAU: 'cross-over point' when becomes useful (typically key size > 2KB)
- SPU has (per core, per algorithm) control word virtual queues for requests



T2's Crypto Units: Design

- MAU supports modular exponentiation and multiplication (RSA and DSA)
- SPU supports cryptographic operations: DES, AES, SHA1
- can operate in `||`; use `CYRPTO ioctls` to access (NCP and N2CP) device drivers
- in turn require hypercalls to access per-core queues to special memory-mapped I/O addresses
- detailed architecture (courtesy Sun Wiki)



T2's Crypto Accelerators: Comments on Design

- SPU, MAU accessible only through hypervisor:
 - copies requests from the virtual queue to the 'real' queue
 - all logical domains can use (but not share) accelerators
- all requests are asynchronous (interrupt based)
- design provides transparent (opaque?) access; parallel operation enabled by kernel
 - intended to have minimal impact on core's pipeline and hardware threads
 - nonetheless large overheads (OK only for large keys)
 - was obscurity an objective? was on-chip accelerators a good tradeoff?
- performance: typically 30-50 Gb/s for 'bulk' operations, $\approx 10\times$ that for 4-core 2.7 GHz x64 processor
 - T2 cores have significant time for other processing
- references: see papers & links so far, and also Cody's COMP3130 Report
 - background on cryptography; speedups for multithreaded requests