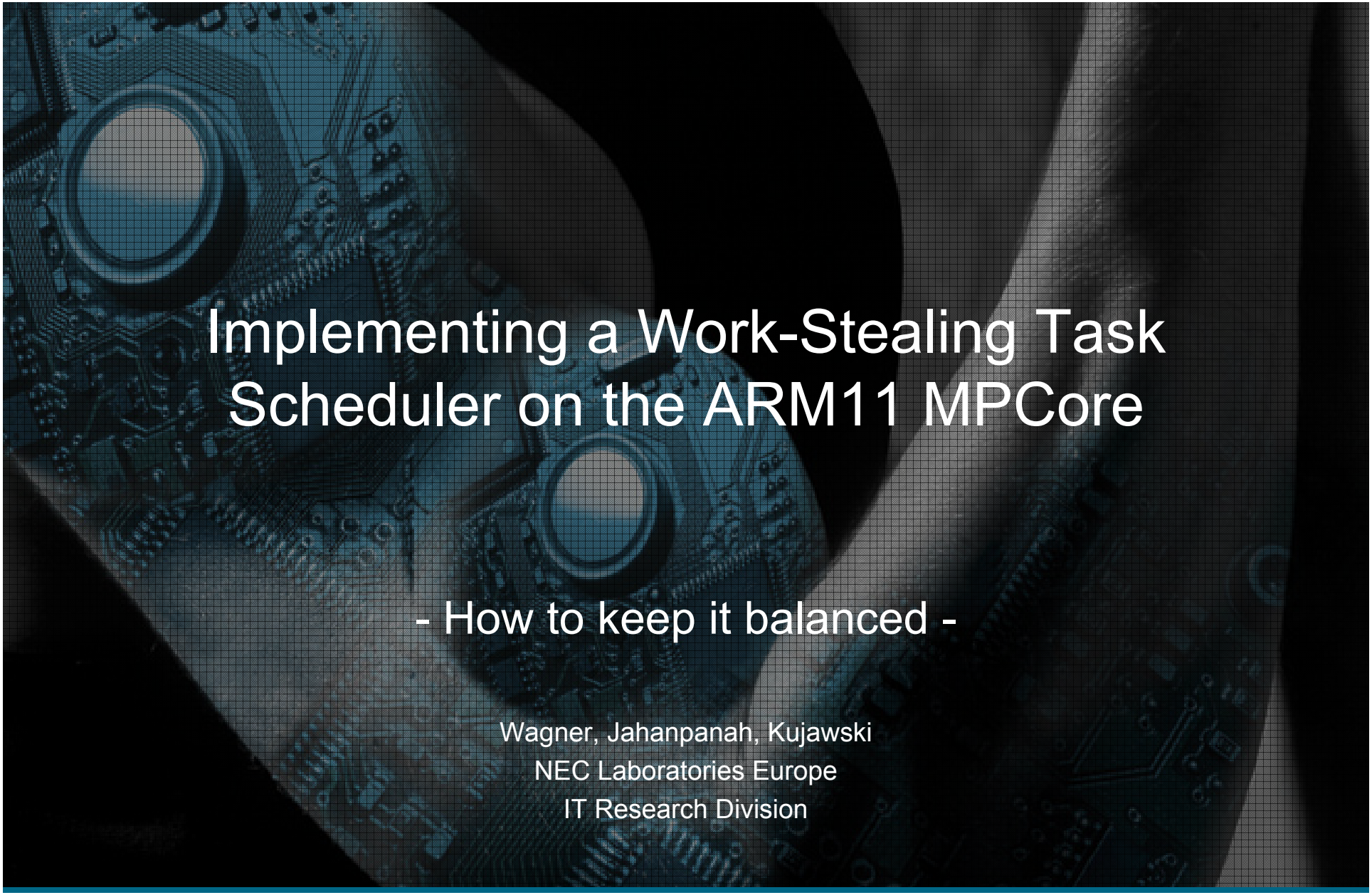


Implementing a Work-Stealing Task Scheduler on the ARM11 MPCore

- How to keep it balanced -

Wagner, Jahanpanah, Kujawski
NEC Laboratories Europe
IT Research Division

Content

1. Introduction and Motivation
2. Parallel Concepts and related Background
3. TPI – a Task Processing Interface
4. Landscape Rendering
5. TPI - Code examples
6. Results and Outlook

1

Herb Sutter:

“THE FREE LUNCH IS OVER”

(30th March ’05, Dr. Dobb’s)

1 Multicores are coming

- MPCore Test Chip was available since May 2004
  - Slow acceptance (because of no free lunch ?)
- Multicore “revolution”
  - Sequential software does not exploit the new architecture
  - Automatic parallelization does (still) not work well

    ⇨ rewrite code for parallel execution:
    1. Thread based static parallelization
    2. Fine-granular task based parallelization
    3. Language change (from C/C++ to a “parallel” language)
    4. MPI / OpenMP
- Code needs to get touched anyway

  ⇨ change it in a way that it scales with any number of cores (if problem allows that)

1 Divergent HW

- Multicore architecture changes
  - N General purpose CPUs + X specialized cores
    - Cell: PPC + SPUs
    - X86: CPU + GPU
      - AMD's Fusion concept
    - Sun T2
    - …
  - Shared memory vs. distributed memory
    - SM is wonderful from an algorithmic point of view
      - Difficulties in coherence and access bandwidth when increasing N
    - DM is easier to maintain but puts another burden on the user

? Is there a programming concept which fit's on most architectures ?

2 Concepts

- Static parallelization
  - Divide problem's domain into subdomains matching the number of cores
- But:
  - Number of available cores might change at runtime
    - High-Priority apps might interrupt us
  - Time required to compute each partition might be different
    - What's the proper domain of dividing the problem ?
    - Is the needed amount of time known in advance ?
    - Dynamically adjustment of partition sizes needed for good load balancing !

2 Concepts

- Solution: Generate much more threads than processors available
  - Each thread should do less work
  - OS'es scheduler will ensure proper balancing without the need of runtime partitioning adjustments

- But:
  - Who will implement the complicated synchronization ?
    - Imagine usually a thread count > 100 for fine granularity of the problem
  - Scheduling overhead (in the OS) grows
  - Synchronization costs are getting higher
  - Lock-based synchronization suffers by swapped out threads (if they hold the lock)

2 Concepts

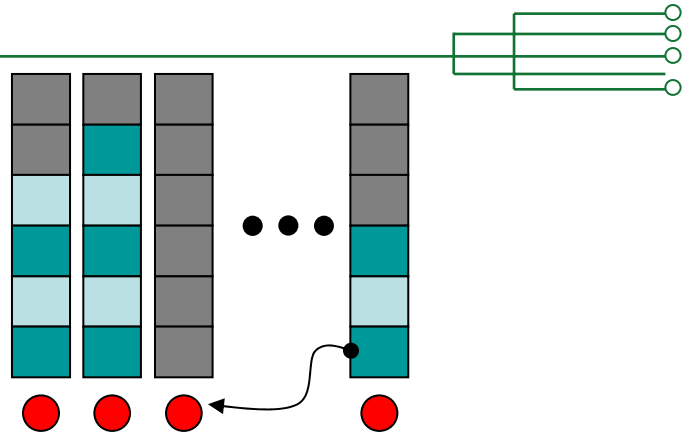
- Schedule “atomic tasks” instead of threads using a user-space scheduler
- Don’t have the OS do things you can do more efficient because of deeper knowledge of the problem
  - A task is an elementary working unit not meant to get interrupted just to serve another task of the same problem
    - Tasks don’t compete against others (like threads do)
    - Very simple scheduling
  - A task should be created only when the problem needs it
    - No waiting on lock-based conditions (blocking)
- Restricted ways of communication between tasks
  - Work, don’t chat !
  - Only provide tree-like dependencies in data flow
  - Try handling other dependencies implicitly “just-in-time”

2 Background

- Idea of User-Space Task Scheduler
  - Task queues are considered at least since ’81 (Burton, Sleep)
    - One big queue serves all nodes
  - Node-local queues since ’91
    - Lowers the synchronization overhead
  - Randomized work stealing ’94 (Blumofe, Leiserson)
    - If a node runs out of tasks, pick a random neighbor's next one
  - Non-blocking work stealing (Blumofe, Papadopoulus) ’98
    - Each queue runs fully independent of the neighbor’s queue’s state
  - Migration-avoiding work stealing (Acar, Blelloch, Blumofe) ’00
    - Optimized scheduling to exploit cache locality

2 Background

- Implementing a custom scheduler
  - Ingredients from Theory
    - Lock free queues
    - Worker thread(s) per local queue



  - Embedded system restrictions (the Engineer's Compromise)
    - Cooperativity and power consumption
      - we are not the only application
      - busy-waiting vs. sleeping in the idle case
    - We don't have N=1000nds of cores (yet)
      - O(log N) vs. O(N), how big is "c" for current architectures ?
      - Maintain option to switch to complicated O(log N) if time is right

2 Background

- Locks are not optimal if
  - Frequent concurrent access to data structures is needed
  - More threads than #processors need to get synchronized

  ⇨If the lock holder is swapped out, all others have to wait
  - Locks are elements of Dead-Locks

- Lock free techniques circumvent that but have
  - Slightly higher costs for the ideal execution time
    - Test and Set vs. complicated program flow
  - Exploding numbers of possible program states
    - Problem for state-based certification schemes ?
  - Theoretical occurrence of Life-Locks in certain designs
  - Complicated and dedicated algorithms for every basic data structure

2 Background

- Use published and tested standard algorithms for lock free
  - LIFOs
  - FIFOs
  - Linked lists, skip-lists, dictionaries
  - Memory management

- Build everything on top of lock free atomic primitives
  - TestAndSet
  - AtomicAdd / AtomicSub
  - CompareAndSwap / LLSC

Universal wait-free constructs [Herlihy '91]

- Still complicated to get it always right

2 Background

- On ARMv6 use ldrex / strex combination for implementing atomic primitives

```
retryTnS:
 __asm {
  ldrex oldval,[addr]
  strex tmp,one,[addr]
  teq tmp,#0
  bne retryTnS }
```

ABA safe

Life lock free for r0p2 rev.

- Beware of architecture specific behaviours
  - Read/Write re-ordering might happen (store buffers)
    - Use proper “acquire”/”release” semantics for signaling atomic operations using CP15’s Data Memory Barrier command on ARMv6

```
__asm { mcr p15,0,0,c7,c10,5 }
```

2 Background

- How do the caches behave ?
  - SCU takes care of L1-sync'ing on MPCore
  - Other architectures may need explicit syncs

- Have an eye on the compiler
  - Instruction scheduling barriers
    - `__scheduler_barrier()` for armcc
    - `__asm__ __volatile__("" : : : "memory")` for gcc
  - "volatile" declared variables
    - (Hopefully) prevent compiler from doing aggressive optimizations
      - ⇨ Important for loop counters
    - Check asm output if switching to a different tool chain

2 Background

- Re-using Hans Boehm's atomic_ops library:
  - Supports commonly used atomic operations
  - Part of the famous Boehm-Demers-Weiser GC for C/C++
  - It already comes with x86, x86-64, Sparc, IA64, …
  - Works with icc, gcc, msvc,…
  - Linux, Windows,…
  - Open Source
  - Optional: lock-based emulation for error-checking

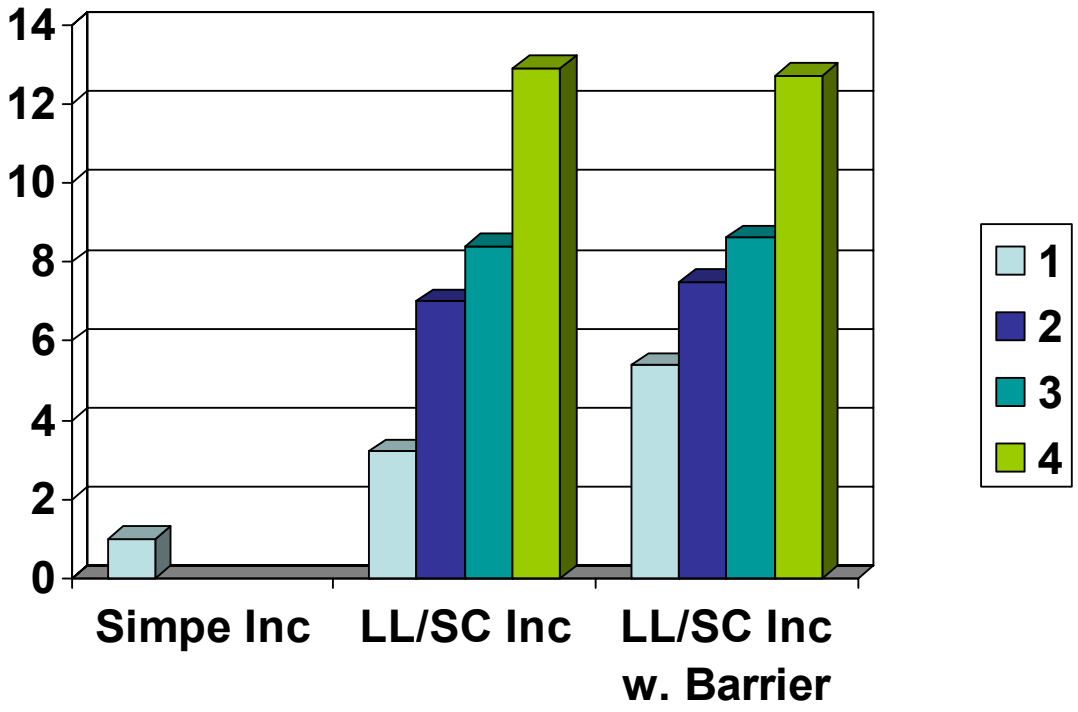
- Extended it with paths for ARMv6 using gcc or armcc
  - Patch is currently reviewed for integration

2 Background

- Relative timings for atomic increment vs. local increment (1.0)

⇨ Expect factor 5 in weak concurrent scenarios when having it combined with a full barrier (2x DMB)

⇨ Expect atomic operations become a serialization point in high concurrent scenarios



2 Background

- Testing on different multi-core platforms faster reveals bugs

- ARM11 MPCore on EB is not the fastest development environment (yet)
  - NEC : 700 – 900 MHz ARM11 MPCore ready before end of 2007
  - EB board provides r0p0 silicon running at 175 MHz

⇨We implemented everything on Dual/Quad-Core x86 machines first

3 TPI – Task Processing Interface

- Built to handle tasks via a small and neat interface
  - **Register** tasks
    - Tasks can have a ‘finalizer’ assigned
    - Finalizer gets enqueued when all children are finished
  - **Enqueue** tasks
    - Either from external (non-worker) thread or from within TPI
  - **Wait** for tasks
    - Blocking wait from external threads for enqueued tasks
  - **Abort** tasks
    - Asynchronously cancel tasks (and their children)
- Browser based inspection interface
  - Simulate variable number of CPUs at runtime of the application
  - Graphical overview about queue’s contents
- Cross-platform (x86, x86-64, ARM, Linux, Windows)

3 TPI – Internals

- Local FIFO per worker thread storing pending tasks
- New tasks are stored in the creator's queue
- External Tasks are put into a separate queue
  - External ⇔ Spawned from non-worker threads
- Work stealing uses a random permutation when checking queues
  - Prevents convoys when several threads are stealing work
- Worker threads fall asleep if stealing failed
  - No further stealing attempts after one circular round trip over all queues

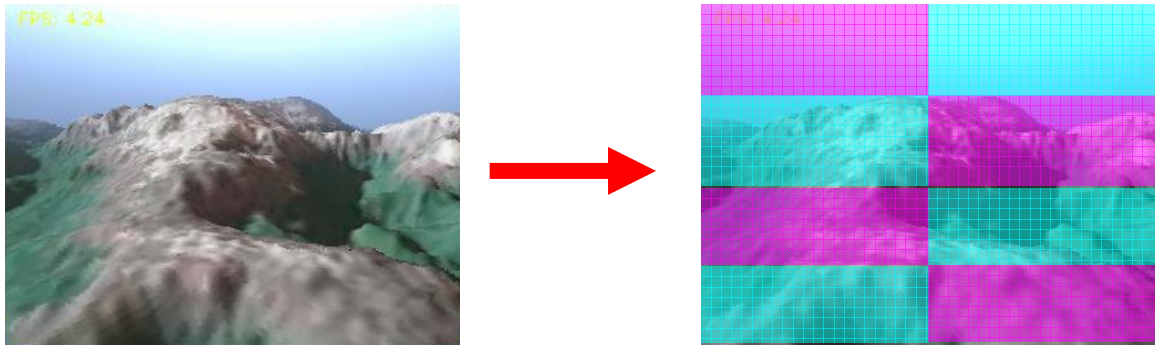
3 TPI – Internals

- Pending tasks are identified via handles allowing
  - Asynchronous aborts
  - Blocking waiting on finishing
  - Reference counting
  - Changing the finalizer from within a running task
- Each task only knows it's parent
  - Master tasks have no parent
- Each task counts the number of "pending" children
  - Easy way to trigger a finalizer call as soon as the last child has finished execution
- Finalizers may get executed
  - Instantaneously
  - As child task again
  - As independent sibling task

⇨ Finalizers act like a barrier waiting for all children

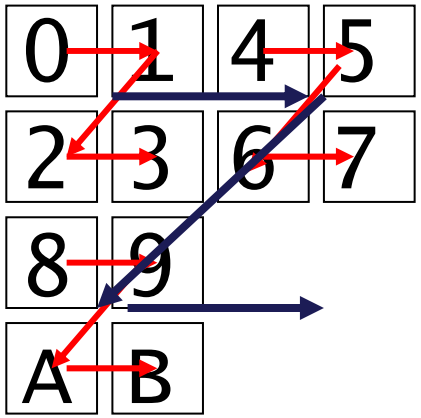
4 Raytracing a Landscape using TPI

- IBM's demo on CELL – what can be achieved on MPCore ?
  - 3.2 GHz vs. 175 MHz (on ARM EB)
  - CPU Design 2:8 vs. 4:0
  - Current EB has **slooow** memory connection
  - MPCore has no floating point SIMD (VMX/SSE)
  - ⇨ HD 1080p vs. QVGA
- Landscape given by height map and texture
- 6 DOF camera
- Partitioning scheme: Fixed in screen space



4 Raytracing a Landscape using TPI

- 1st optimize inner loops
  - VFP => fixed point integer
  - layout source data for optimal cache usage
    - Morton order pattern for heightmap and texture data

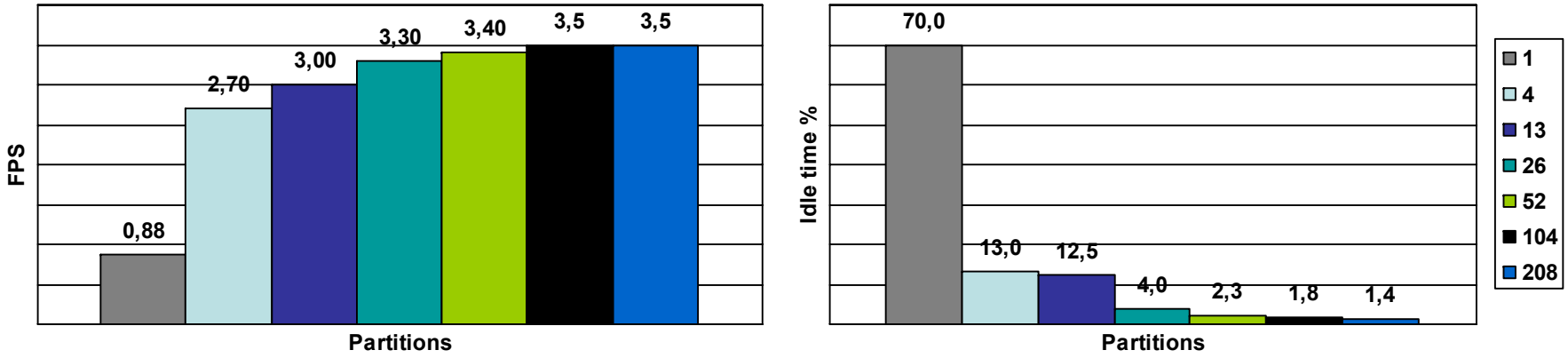


  - Use hierarchical collision tests (quick skip)
    - 16 x 16 blocks

⇨ Totally memory bound

4 Raytracing a Landscape using TPI

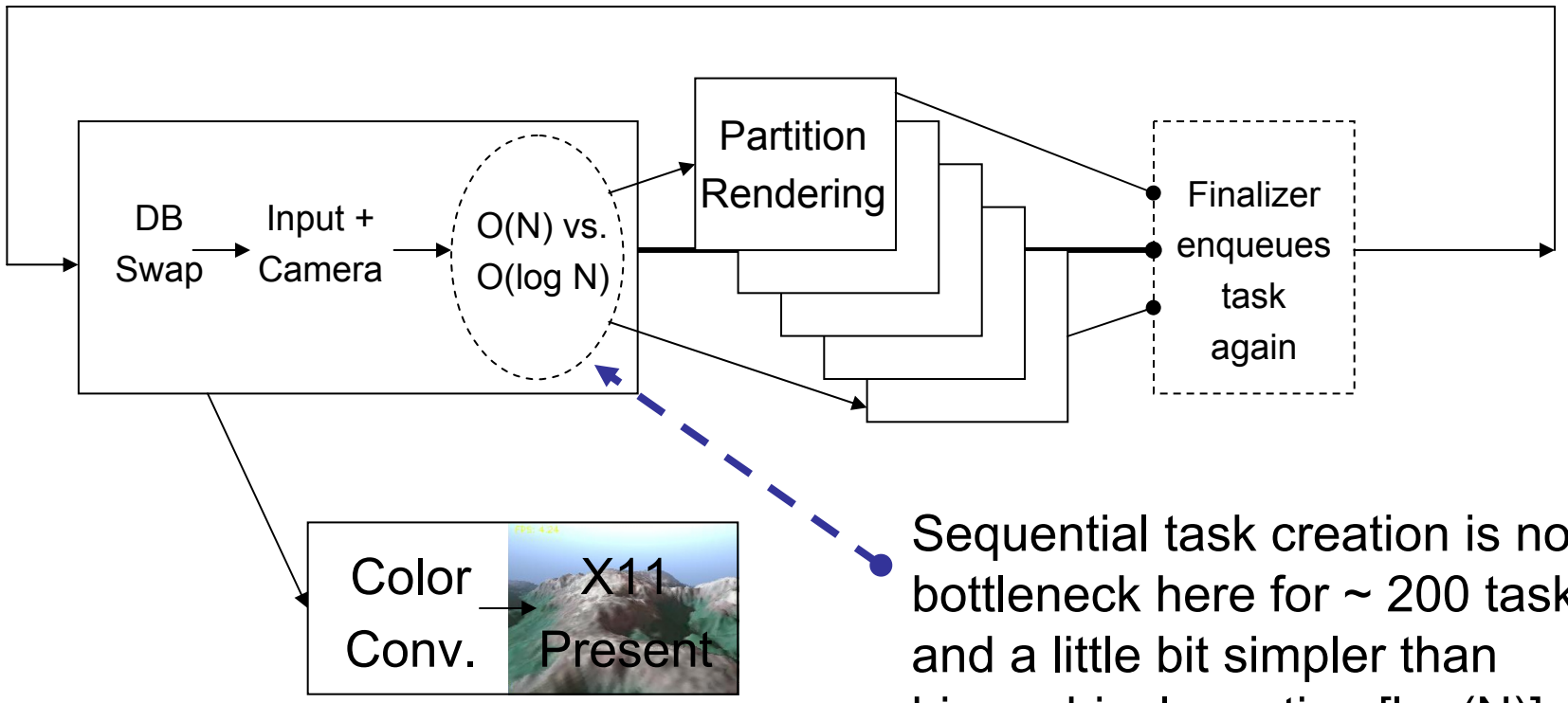
- 2nd parallelize everything possible via task scheme
  - Double-Buffering
    - X-Window update only uses one core
    - Without DB we would have to block here having 3 cores idle
  - Rendering into partitions



  - TPI overhead with 208 tasks: <0.2 % (measured via oprofile)

5 TPI Code examples

- Program flow diagram at task level



Sequential task creation is not a bottleneck here for ~ 200 tasks and a little bit simpler than hierarchical creation [log(N)]

5 TPI Code examples

```
// MAIN thread
TPI *tpi = TPI::Create();

frameTaskID = tpi->Register(frameTask,TASKID_SELF);
traceTaskID = tpi->Register(traceTask,TASKID_NONE);
presentTaskID = tpi->Register(graphPresentTask);

tpi->extEnqueue(frameTaskID);

while(!input.m_shouldQuit)
  input.Update(0, true);

gMustQuit = true;
tpi->extWaitAllFinished();
tpi->Destroy();
```

5 TPI Code examples

```
TaskParam* frameTask(TaskParam* parameter)
{
  graphSwap();
  TPI::EnqueueChild(presentTaskID);
  timer.Advance();
  camera->DoMovement(timer.GetDelta());

  for(int t=0; t < tracePartition::partitionCount; t++)
    TPI::EnqueueChild(traceTaskID, partitions + t);

  if(gMustQuit)
    TPI::GetCurrentTaskDesc()->SetFinalizer(TASKID_NONE);

  return parameter;
}
```

6 Results

- Encountered problems
  - Only 16 bit frame buffer
    - Color conversion must be done via CPU but we have not parallelized it
  - Often gcc is producing faster code than armcc

- Kernel 2.6.22 + glibc 2.5 + r0p0 MPCore
  - Mutual exclusion via pthread mutexes does not work reliably
    - The “why” is still unclear ?
  - r1p0 seems to work reliable in FPGA implementation

  Lesson learned: Write testing code even for the basic things
  - Give it time to crash, multi-core races don’t occur frequently!

6 Results

- Higher level C++ abstractions on top of basic scheduler functions
  - `parallel_for_each<>`
  - `split_and_gather<>`
  - `parallel_while<>`

- In-House test cases using C++ templates for
  - parallel matrix multiplication (`parallel_for_each`)
  - Fibonacci series (`split_and_gather`)

6 Outlook

Existing work stealing alternatives (providing higher level abstractions)

- Cilk-5 Language — http://supertech.lcs.mit.edu/cilk
- Hood scheduler (Blumofe 1998)
- Intel TBB (Open Source now) — www.intel.com/software/products/tbb
- Factory scheduler — http://people.cs.vt.edu/~scschnei/factory
- Mercury MCF for CELL — http://www.mc.com/

OS Functionality (involves kernel overhead)

- Linux: Tasklets, WorkQueues
- Windows: WorkQueues

6 Outlook

- What about a lowest common standard ?
  - Lower level task abstraction and handling only needs a couple of functions
    - Spawning
    - Aborting
    - Simple signaling (e.g. finalizer scheme)
    - Simple barriers
  - If possible don't bind it to specific language
    - Wouldn't a FORTRAN interface be nice for some people ?
  - Build custom higher level algorithms on top of standard
    - e.g. C++ template constructs
  - Serve distributed systems (Clusters) ?
  - Serve heterogeneous systems ?

Closing the circle

Herb Sutter:

"O(N) parallelization is the key to re-enabling the free lunch …"
(3rd August '07, Dr. Dobb's)

But we don't want to have different

dress-code for every party,

a common standard would be preferable.

Q & A

wagner@it.neclab.eu

Literature:

A Simple Load Balancing Scheme for Task Allocation in Parallel Machines (1991) Larry Rudolph

Scheduling Multithreaded Computations by Work Stealing (1994) , Robert D. Blumofe, Charles E. Leiserson

The Performance of Work Stealing in Multiprogrammed Environments (1998) Robert D. Blumofe, Dionisios Papadopoulos

The data locality of work stealing (2000) , Umut A. Acar , Guy E. Blelloch, Robert D. Blumofe