Extreme Specialization

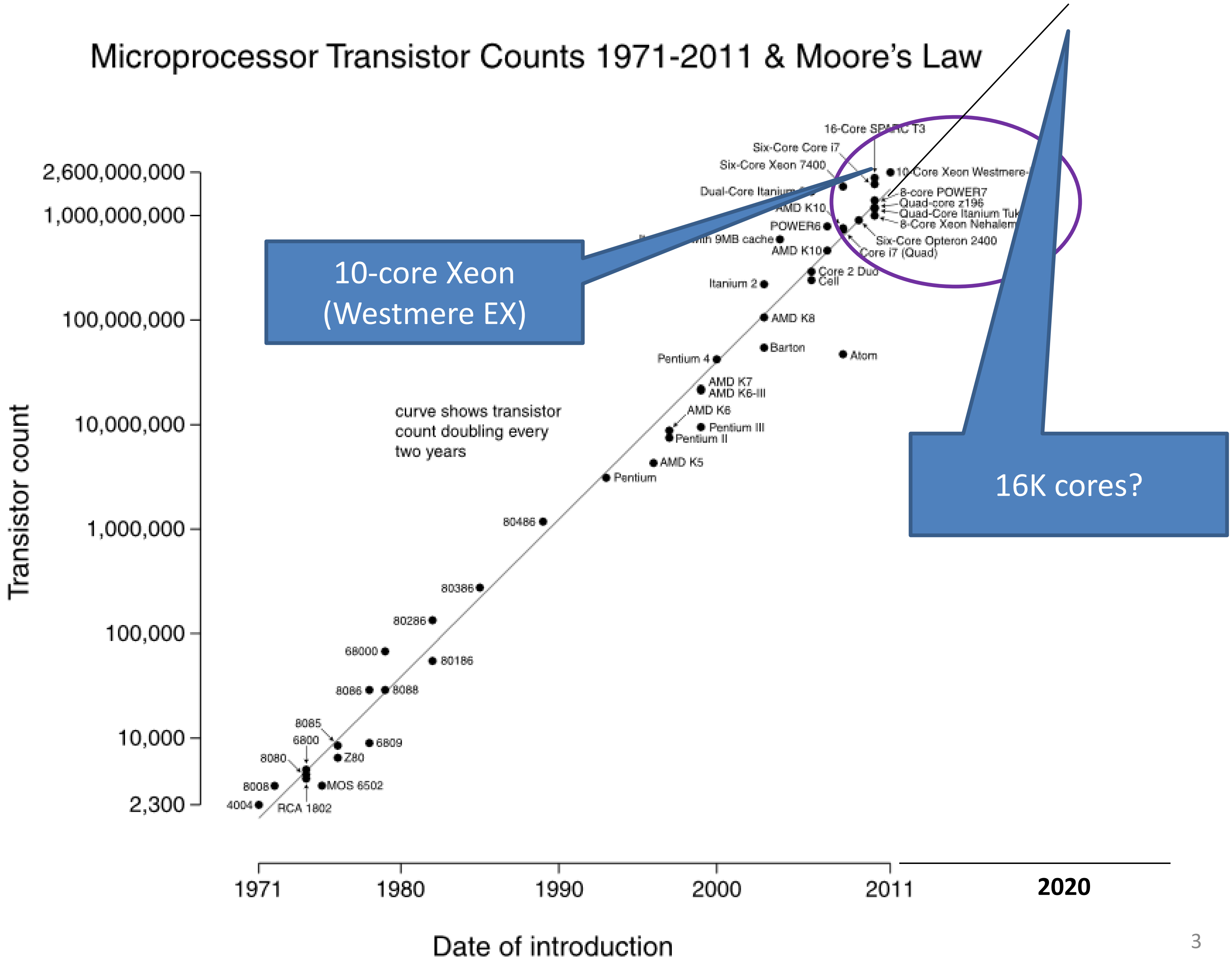
CREST Workshop 22/03/12

Steven Hand

Multicore, Manycore & Mayhem

- The era of M*core is upon us
  - Standard desktop machines now quad core (and standard servers are 2x or 4x this)
  - 8- and 12-core processors around the corner
  - Intel MIC & Tilera & foor & bar & baz => AIEE!!!

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Multicore, Manycore & Mayhem

- The era of M*core is upon us
  - Standard desktop machines now quad core (and standard servers are 2x or 4x this)
  - 8- and 12-core processors around the corner
  - Intel MIC & Tilera & foor & bar & baz => AIEE!!!
- Considerable reaction from academia & industry
  - Moore’s law is dead!
  - We need new paradigms! (or at least new software)
- This talk will cover some of my thoughts on this
  - **Warning**: speculative, argumentative, XXXative – and quite possibly plain wrong!

Is there really a problem?

- Today’s server systems work pretty well
  - HPC and similar – extremely parallel, scale easily
  - Existing server apps – extremely parallel, scale easily
  - OSes fine too – TxLinux (SOSP’07) shows max 12%
  - Brief panic (Corey, OSDI’08)
- Transactional memory not reqd for performance
  - Roy (HotPar’09) shows zero speed up for Apache
  - TxLinux shows 4-8% benefit from HTM (1% for x16!)
- And if they don’t, VMMs (or other strongly partitioned OSes like Barrelfish) provide a decent solution
  - Disco (SOSP’95) was rather prescient…

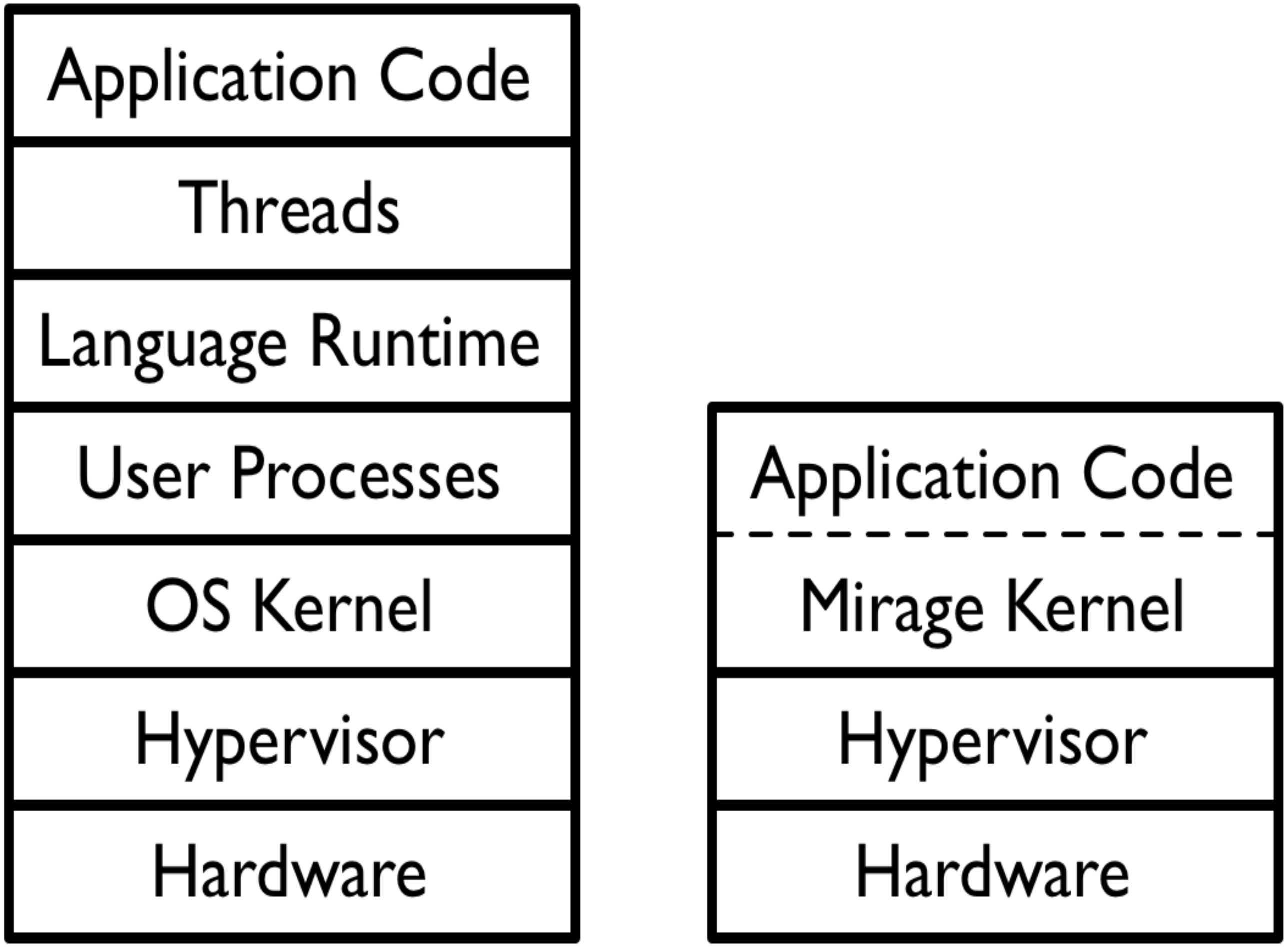
But what about new applications?

- One argument is that (most) programmers just shouldn't worry about ||-ism
  - although, anecdotally, many seem to :-(
- Instead focus on ***strategies*** (like divide and conquer)
  - Or on annotations (OpenMP, *-SS, …)
  - Or on libraries (Intel's TBB, java.util.concurrent, ..)
  - Or on task-parallel programming frameworks (e.g. Cilk or MapReduce/Phoenix or Ciel or …)
- Last can potentially support:
  - **Transparent scaling** (up and down = FT story), and
  - **Code mobility** (desktop, cloud, mobile, GPGPU (?), …)

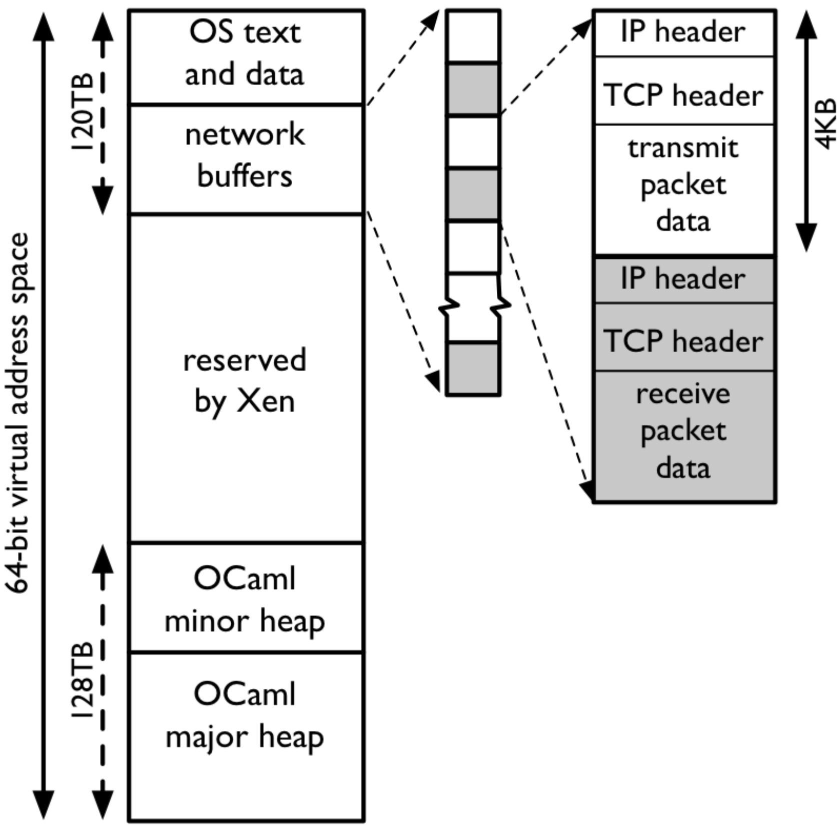
Cloud Run-time Environments

- If we move to new programming paradigms, great potential for scalability and fault tolerance
- But MapReduce/Dryad/Ciel are user-space frameworks in a traditional OS (in a VM!)
  - Do we really need all these layers?
- One possibility is to build a “custom” OS for the cloud (or at least for data intensive computing)
  - E.g. Xen powers most cloud computing platforms
  - It forms a stable virtual hardware interface
  - Therefore, can compile apps directly to Xen “kernels”

MirageOS: Specialized Kernels



MirageOS: Current Design



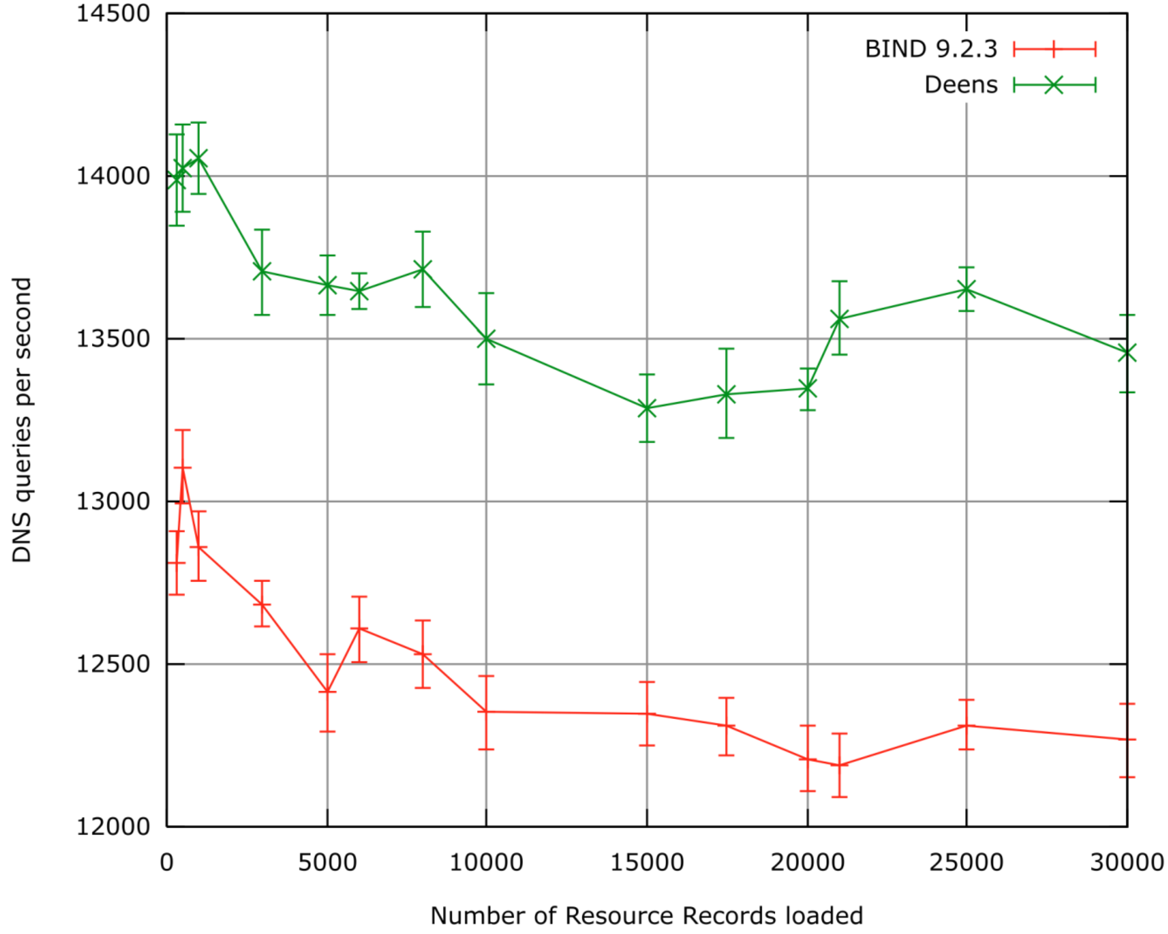
Memory Layout

64-bit para-virtual memory layout
No context switching
Zero-copy I/O to Xen
Super page mappings for heap

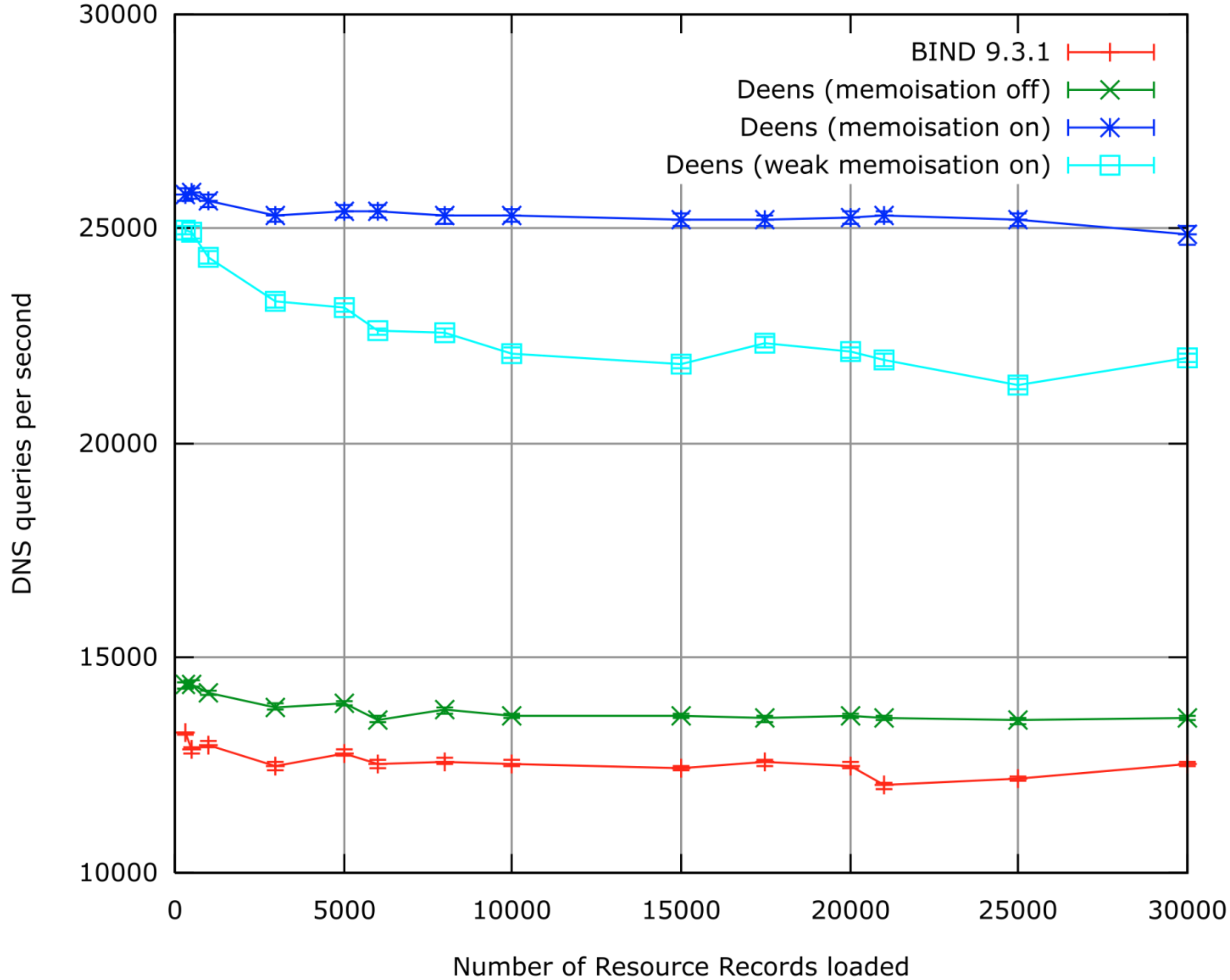
Concurrency

Cooperative threading and events
Fast inter-domain communication
Works across **cores and hosts**

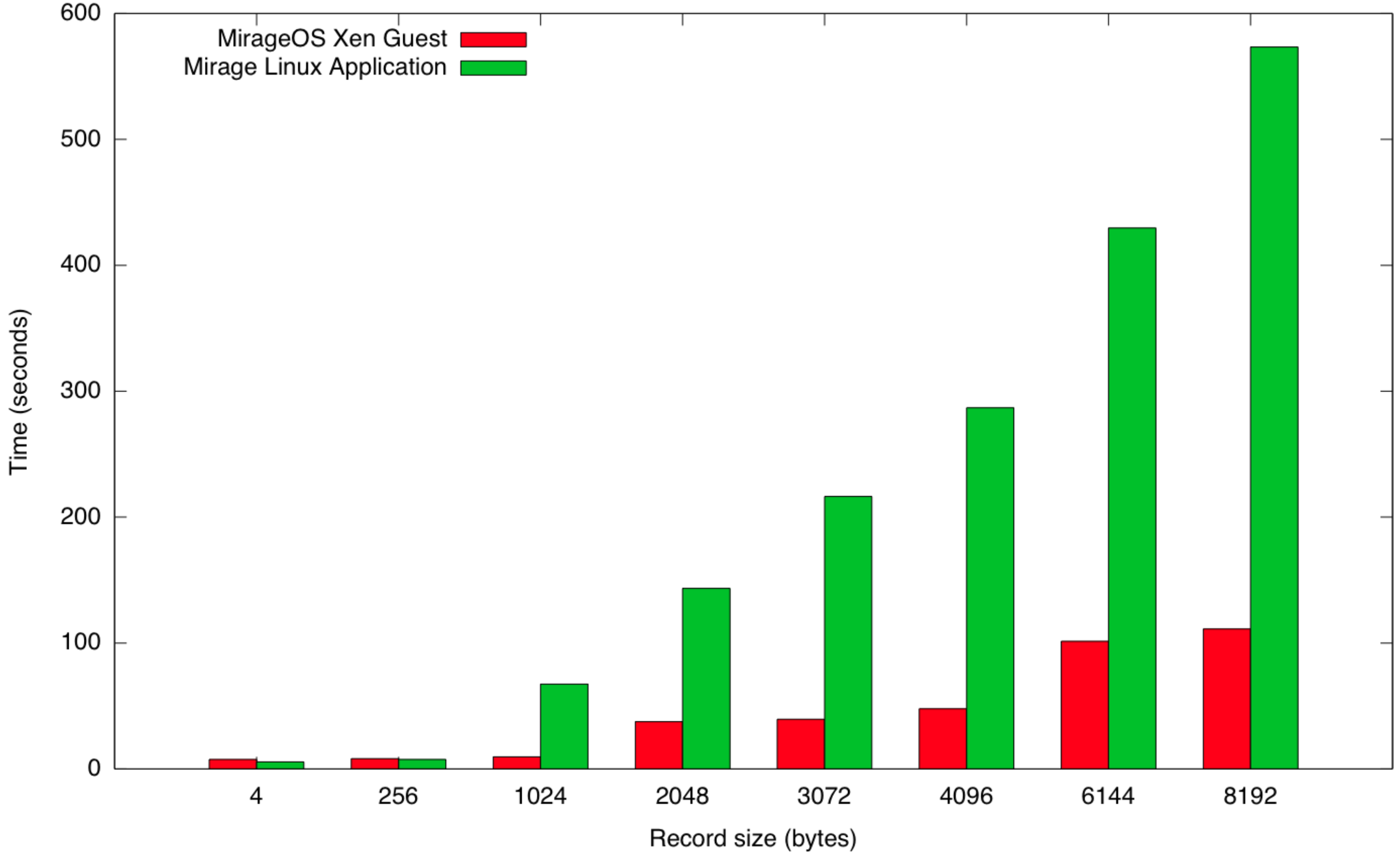
DNS: BIND (C) vs Deens (ML)



DNS: with functional memoisation



SQLite performance vs PV Linux



MirageOS: Status

- Open source, and has self-hosted(!) web site
- Alpha quality code, but under active development at Cambridge & elsewhere
  – Code, tutorial and slides on web site
  – Recent work includes OpenFlow software
  – Supported by EPSRC, Verisign and DARPA

URL: http://openmirage.org/

Peering into The Future

- Unlikely that *everyone* will move to MirageOS and ocaml overnight ;-)
- Q: can we develop tools and systems which help regular programmers to exploit M*-core?
  - not about "auto parallelization" in the traditional sense (i.e. extracting fine-grained parallelism)
  - don't want to make SPECint (or Parsec) faster
- Our focus is on two related strands:
  - semi-automatic transformation of programs into task-parallel / data-flow form (c/f SOAAP), and
  - semi-automatic transformation of single threaded code to exploit additional cores

The Death of Multiprogramming

- Widely overlooked problem with M*-core:
  - What do we do when a thread blocks?
  - Traditional solution (run another thread) doesn’t work so well if very large #cores
- How can we reduce ***wait time*** ?
  - Amount of time ‘the thread’ spends unable to run
- One possibility is **extreme specialization**:
  - Combines ideas from partial evaluation, memoization, dynamic specialization and speculation!

Specializing File I/O

- One student looking at desktop applications
  - e.g. at start of day, load XML configuration file from disk to generate a set of program variables
  - can concretize values at compile stage, and partially evaluate (lots of constant propagation!)
  - can also elide unreachable paths (dead code elimination), and unroll loops, and inline functions
  - can even eliminate threads (or aio) – e.g. for font search paths, plugin scans, etc, etc
- So far seems promising… at least for **start-up**…

Dealing with Uncertainty

- At some stage your analysis breaks down
  - i.e. cannot continue with sound optimizations
- This is an opportunity to **gamble**:
  - Guess which path will be taken (i.e. speculate)
  - Can also speculate on data values
- In vanilla form, this is just symbolic execution
  - Remember the path predicates
  - Generate code guarded appropriately
  - Keep original stuff around just in case
- Have some more extreme run-time options too:
  - e.g. force values into well-behaved ranges (Rinard)

A Use for Many-Core?

- May well be many plausible values with associated paths:
  - Great!
  - Use lots of single-core almost-replicas, each specialized for specific cases
  - Fire up more as and when you encounter more uncertainty (e.g. I/O operations)
  - Garbage collect as needed
  - (Reserve one core for general case if you want ;-)
- System now deterministic in K different universes


© 1997 Slawek Wojtowicz

Wrapping it up

- ||-programming can/should be a specialty
  – don't expect 'regular' programs to do assembly
- Develop a set of useful frameworks/languages
  – Different solutions for different patterns
  – Already made a great start on this
  – Personally expect (hope?) <20 will be enough
- Real challenge is how to use many cores to make life better for the masses
  – app-per-core (or partially evaluated app-per-core) seems like it should work to me
- But then again, I could be wrong ;-)