

“GRASSROOTS ASPLOS”
CAN WE STILL RETHINK
THE HARDWARE/SOFTWARE INTERFACE
IN PROCESSORS?

RAPHAEL ‘KENA’ POSS
UNIVERSITY OF AMSTERDAM, THE NETHERLANDS

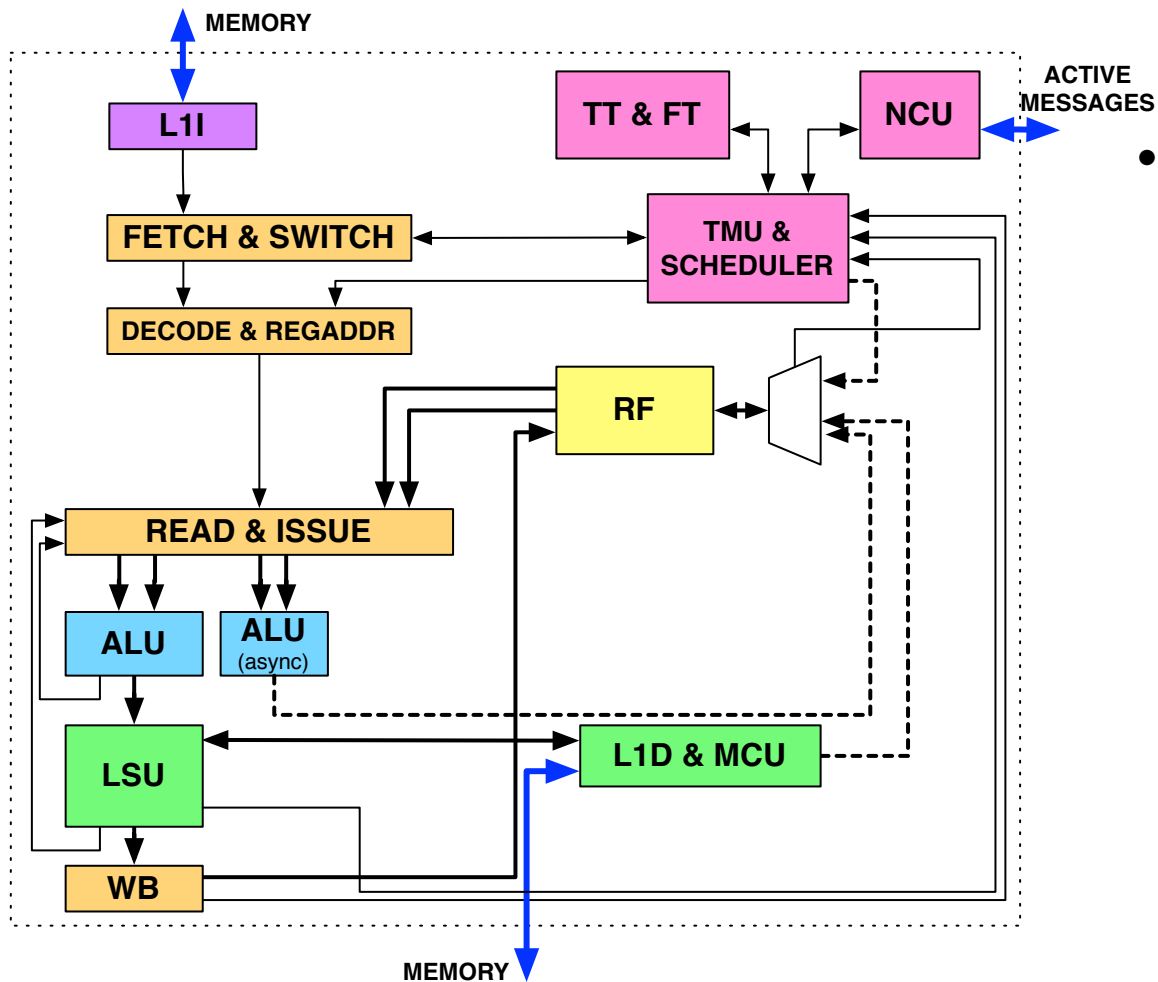
ASPLOS-17 DOCTORAL WORKSHOP
LONDON, MARCH 4TH, 2012



CURRENT ON-CHIP PARALLELISM IS BASED ON LEGACY

- Historical focus on **single-thread performance**
(developments in general-purpose processors: registers, branch prediction, prefetching, out-of-order execution, superscalar issue, trace caches, etc.)
- Legacy heavily **biased towards single threads:**
 - Symptom: **interrupts** are the **only way** to signal asynchronous external events
 - Retro-fitting **hardware multithreading** is **difficult** because of the sequential core's complexity
- **What if...**
we redesigned general-purpose processors,
assuming concurrency is the norm in software?

MICROGRIDS OF D-RISC CORES



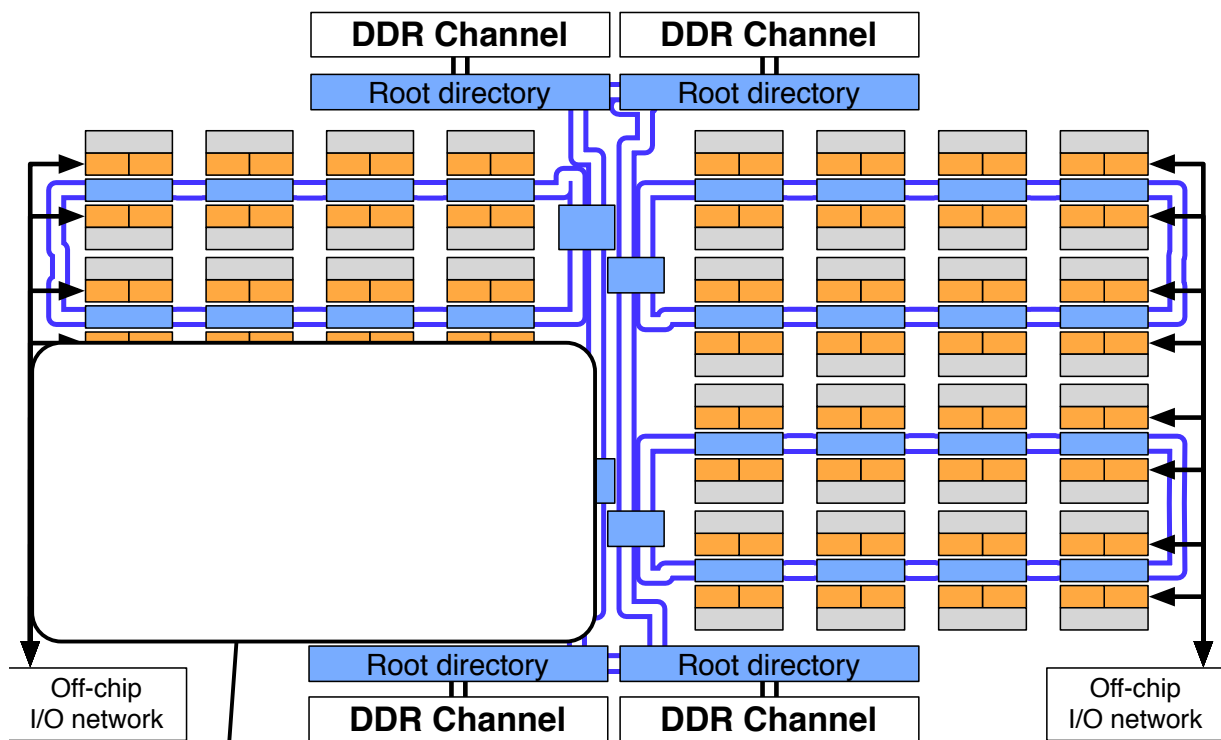
- D-RISC cores:
hardware multithreading + dynamic dataflow scheduling
- **fine-grained threads**: 0-cycle thread switching, <2 cycles creation overhead
- **ISA instructions** for thread management
- dedicated hardware processes for **bulk creation and synchronization**
- **No preemption/interrupts**; events create new threads

In-order, single-issue RISC: small, cheaper, faster/watt

A PERSPECTIVE SHIFT

<p>CORE I7</p>	<p>Function call</p> <p>with 4 registers spilled</p> <p>30-100 cycles</p>	<p>Predictable loop</p> <p>requires branch predictor + cache prefetching to maximize utilization</p> <p>1+ cycles / iteration overhead</p>
<p>D-RISC WITH TMU IN HARDWARE</p>	<p>Bulk thread creation</p> <p>of 1 thread, 31 "fresh" registers</p> <p>~15 cycles (7c sync, ~8c async)</p>	<p>Thread family</p> <p>1 thread / "iteration" reuses common TMU and pipeline no BP nor prefetch needed</p> <p>0+ iteration overhead</p>

EXAMPLE 128-CORE MICROGRID



Approximate size of one Nehalem (i7) core
for comparison

Area estimates with CACTI: 100mm² @ 35nm

- 32000+ hw threads
- 5MB distributed cache
- shared MMU
= single virtual
address space,
protection using
capabilities
- Weak cache coherency
- no support for global
memory atomics –
instead
synchronization using
remote register writes

CONTRIBUTIONS

- 1. Full-system architecture design + description**
uses “companion processor” for legacy OS code
= accelerator model inverted (akin to service nodes in the XMT)
- 2. GNU C compiler + some C library + some POSIX**
(was able to port bits from FreeBSD)
- 3. SL = new C primitives for declarative concurrency**
 - cannot use pthreads / nanox / qthreads / C1x
cost(function call for API) >> cost(ISA thread creation)
 - declarative = *can* be run concurrent, *may* run sequential,
architecture decides based on run-time resource availability

RESULTS, WHAT'S NEXT?

- ✓ **built enough infrastructure to fit the F/OSS landscape**
 - yet can't reuse most existing OS code: *no interrupts, no traps*
- ✓ **as planned, higher performance per area and per watt**
 - via hand-coded benchmarks: *granularity in SPEC is too coarse*
- **Follow-up research areas:**
 - *Internal* issues: memory consistency, scalable cache protocols, ISA semantics, etc.
 - *External* issues from outside architecture: how to virtualize? how to place tasks over so many “workers”? how to port existing OS code?
 - *Fundamental* issues: concurrent complexity theory?

CORE ISSUES

- **Is there still room in our ecosystem to rethink the fundamentals?**
- Assuming so, how to enthuse a community in this direction? How to gain traction? How to get funding for manufacturing?
- Suggestions & comments welcome

EXTRA - KEY CONCEPTS

- **Single-issue, in-order RISC** for more performance/watt
- ILP for **latency tolerance**, concurrency sourced from **many in-order threads** per core instead of out-of-order execution of 1 thread
- **Dynamic dataflow scheduling** over a **synchronizing register file** instead of reservation stations / Tomasulo / reorder buffers
- **Avoid speculation** as it is energy inefficient; loops and branch prediction can be replaced by **interleaved dependent threads**
- **Hardware concurrency management**: thread creation, termination, synchronization, communication via dedicated components

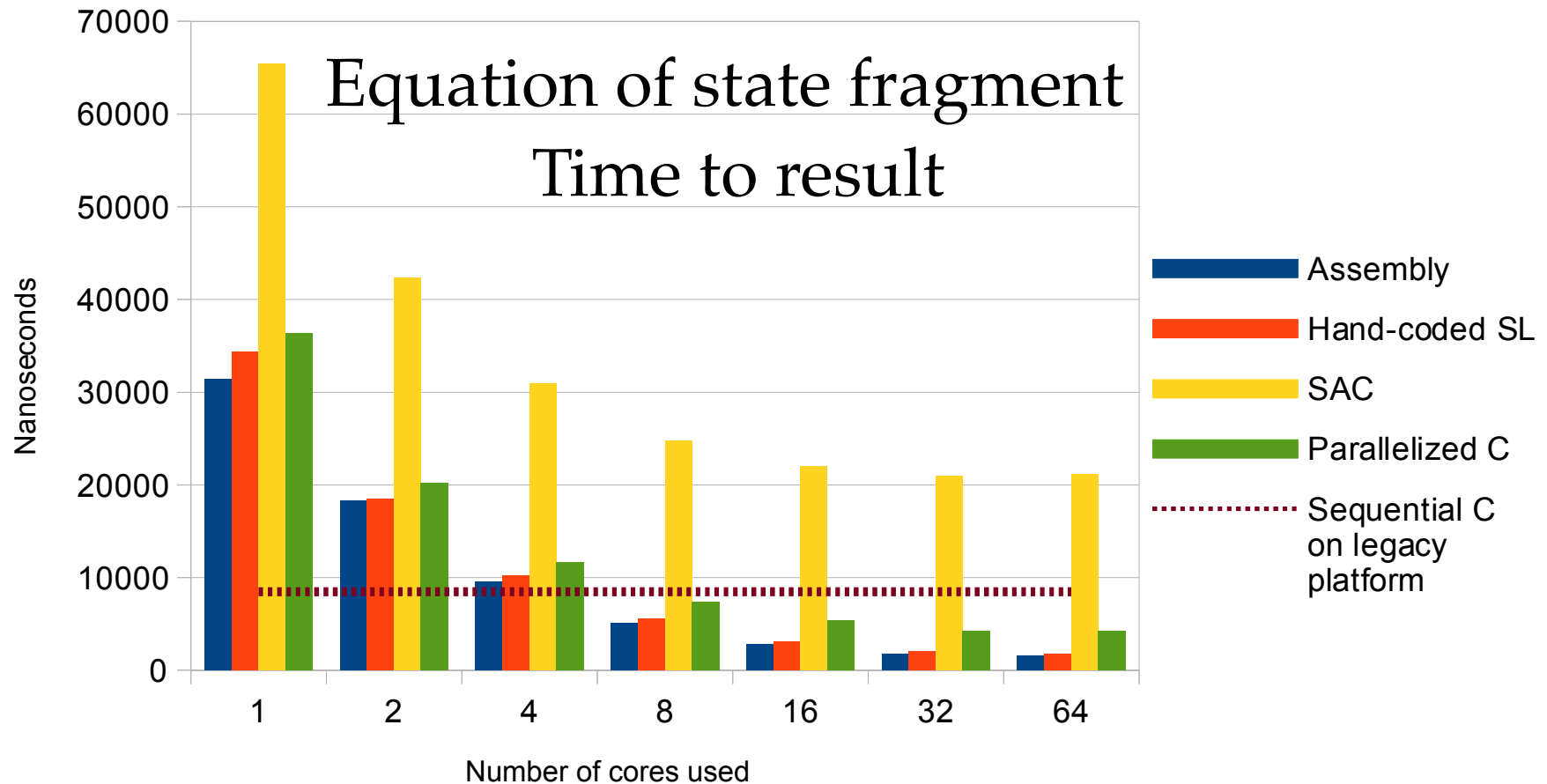
EXTRA - CONCURRENCY MANAGEMENT PROTOCOL

allocate $\$Place \rightarrow \F	Allocate a family context
setstart/setlimit/ setstep/setblock $\$F, \$V \rightarrow \emptyset$	Prepare family creation
create $\$F, \$PC \rightarrow \$ack$	Start bulk creation of threads
rput $\$F, R, \$V \rightarrow \emptyset$ rget $\$F, R \rightarrow \V	Read/write dataflow channels remotely
sync $\$F \rightarrow \ack	Bulk synchronize on termination
release $\$F \rightarrow \emptyset$	De-allocate a family context

EXTRA - A PERSPECTIVE SHIFT

<p>CORE I7 LINUX</p>	<p>Thread creation (pre-allocated stack) >10000 cycles in pipeline</p>	<p>Context switch syscalls, thread switch, trap, interrupt >10000 cycles in pipeline</p>	<p>Thread cleanup >10000 cycles in pipeline</p>
<p>D-RISC WITH TMU IN HARDWARE</p>	<p>Bulk creation (metadata allocation for N threads) ~15 cycles (7c sync, ~8c async) Thread creation 1 cycle, async</p>	<p>Context switch at every waiting instruction, also I/O events <1 cycles</p>	<p>Thread cleanup 1 cycle, async Bulk synchronizer cleanup 2 cycles, async</p>

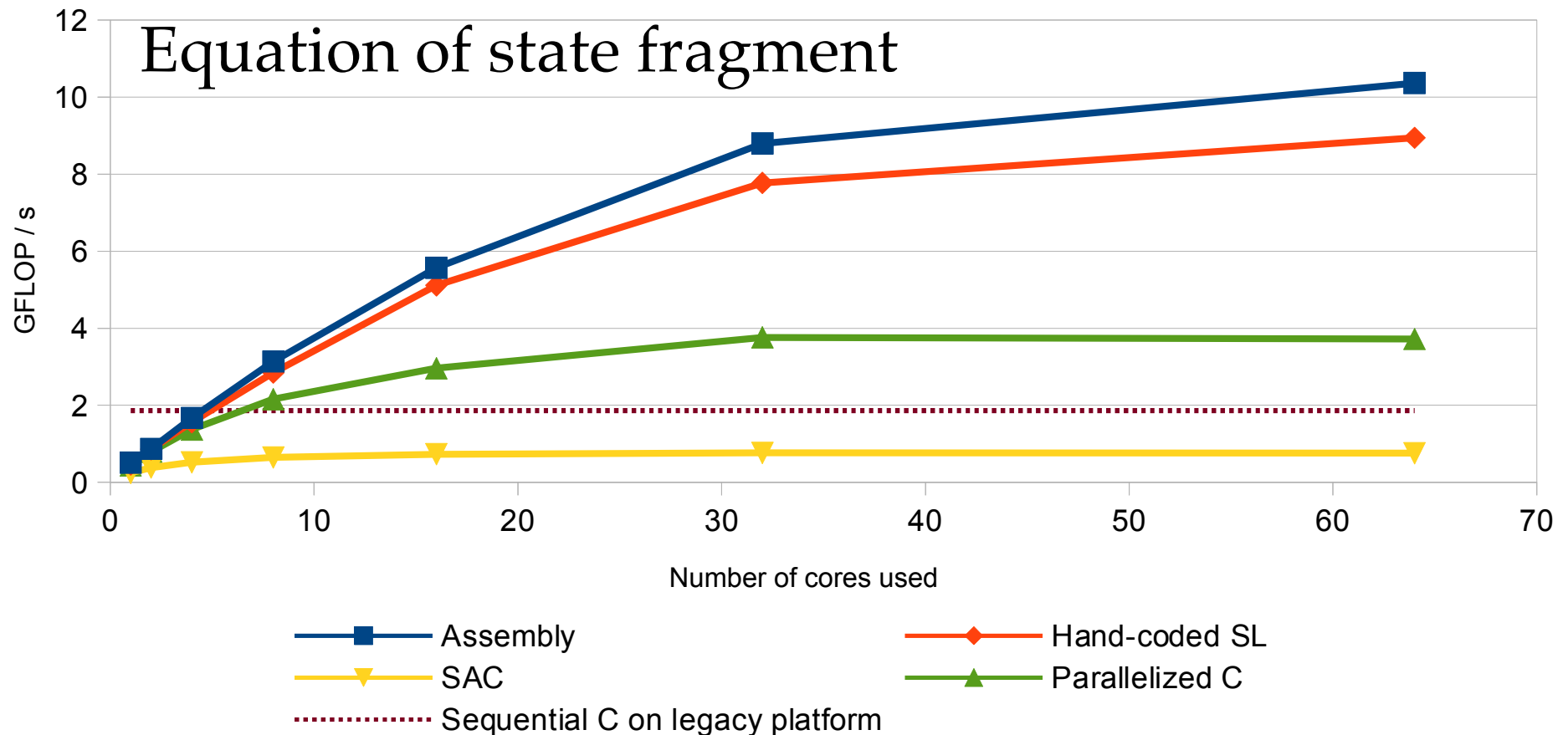
EXTRA - RESULTS



Legacy platform = MacBook Pro, Core 2 Duo @ 2.4GHz

area(1 Core 2 Duo core) ~ area(32 Microgrid cores)

EXTRA - RESULTS



Legacy platform = MacBook Pro, Core 2 Duo @ 2.4GHz

area(1 Core 2 Duo core) ~ area(32 Microgrid cores)