"Grassroots ASPLOS"

can we still rethink the hardware/software interface in processors?

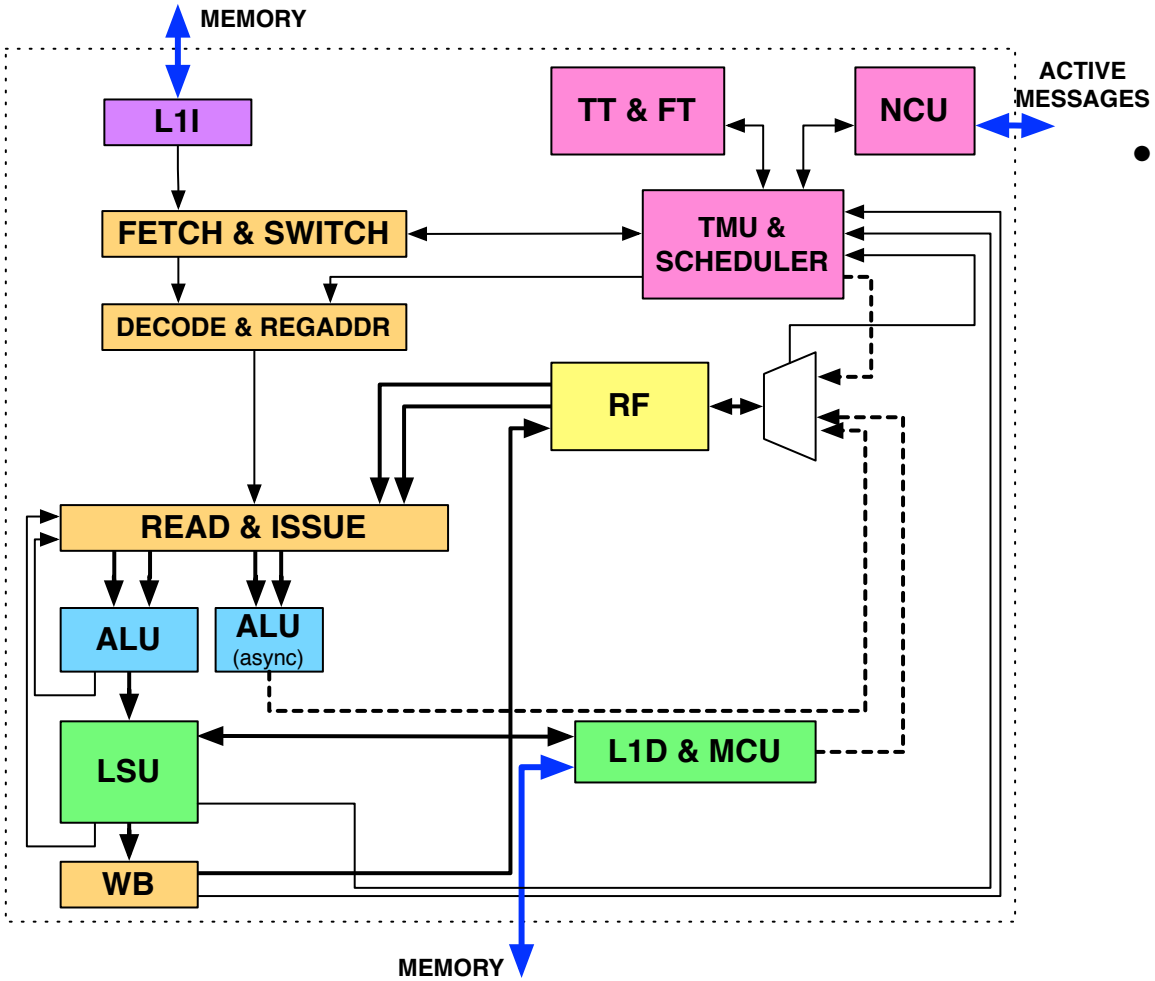
Raphael 'kena' Poss
University of Amsterdam, the Netherlands

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



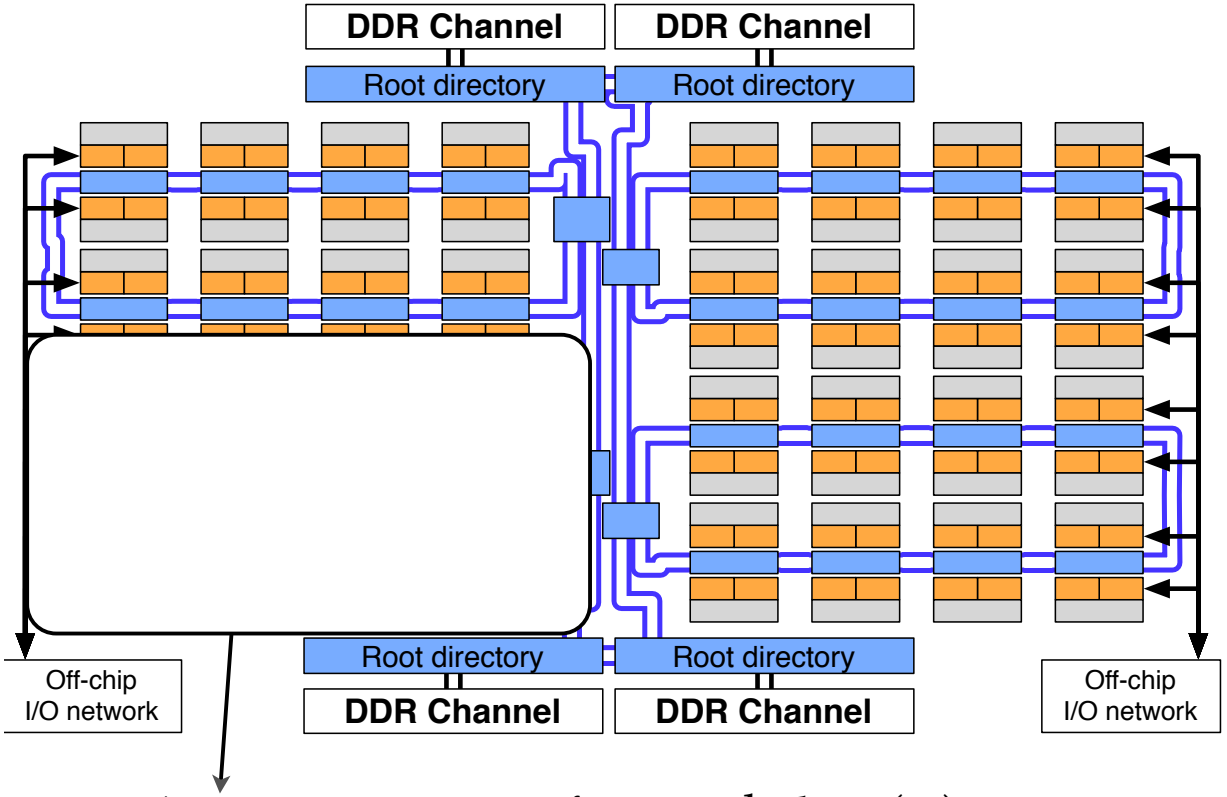
- D-RISC cores:
  hardware multithreading + **dynamic dataflow scheduling**
  - **fine-grained threads**: 0-cycle thread switching, <2 cycles creation overhead
  - **ISA instructions** for thread management
  - dedicated hardware processes for **bulk creation and synchronization**
  - **No preemption/interrupts;**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

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

A PERSPECTIVE SHIFT

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

Example 128-core Microgrid



Approximate size of one Nehalem (i7) core for comparison

- **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

Area estimates with CACTI: 100mm2 @ 35nm

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

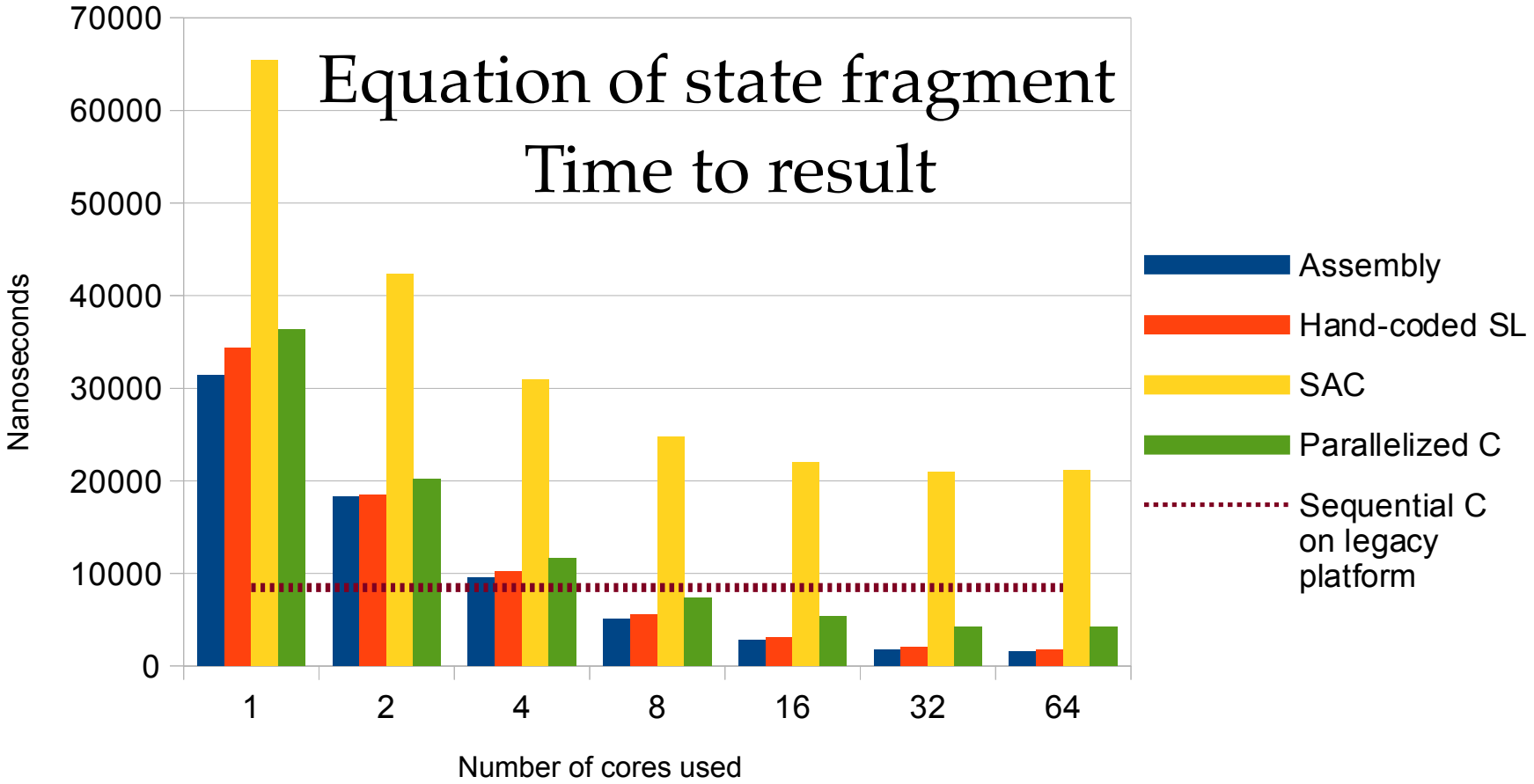
Extra - Concurrency management protocol

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

Extra - A perspective shift

	Thread creation	Context switch	Thread cleanup
Core i7 Linux	(pre-allocated stack) **>10000 cycles in pipeline**	syscalls, thread switch, trap, interrupt **>10000 cycles in pipeline**	**>10000 cycles in pipeline**
D-RISC with TMU in hardware	**Bulk creation** (metadata allocation for N threads) **~15 cycles** **(7c sync, ~8c async)** **Thread creation 1 cycle, async**	**Context switch** at every waiting instruction, also I/O events **<1 cycles**	**Thread cleanup 1 cycle, async** **Bulk synchronizer cleanup 2 cycles, async**

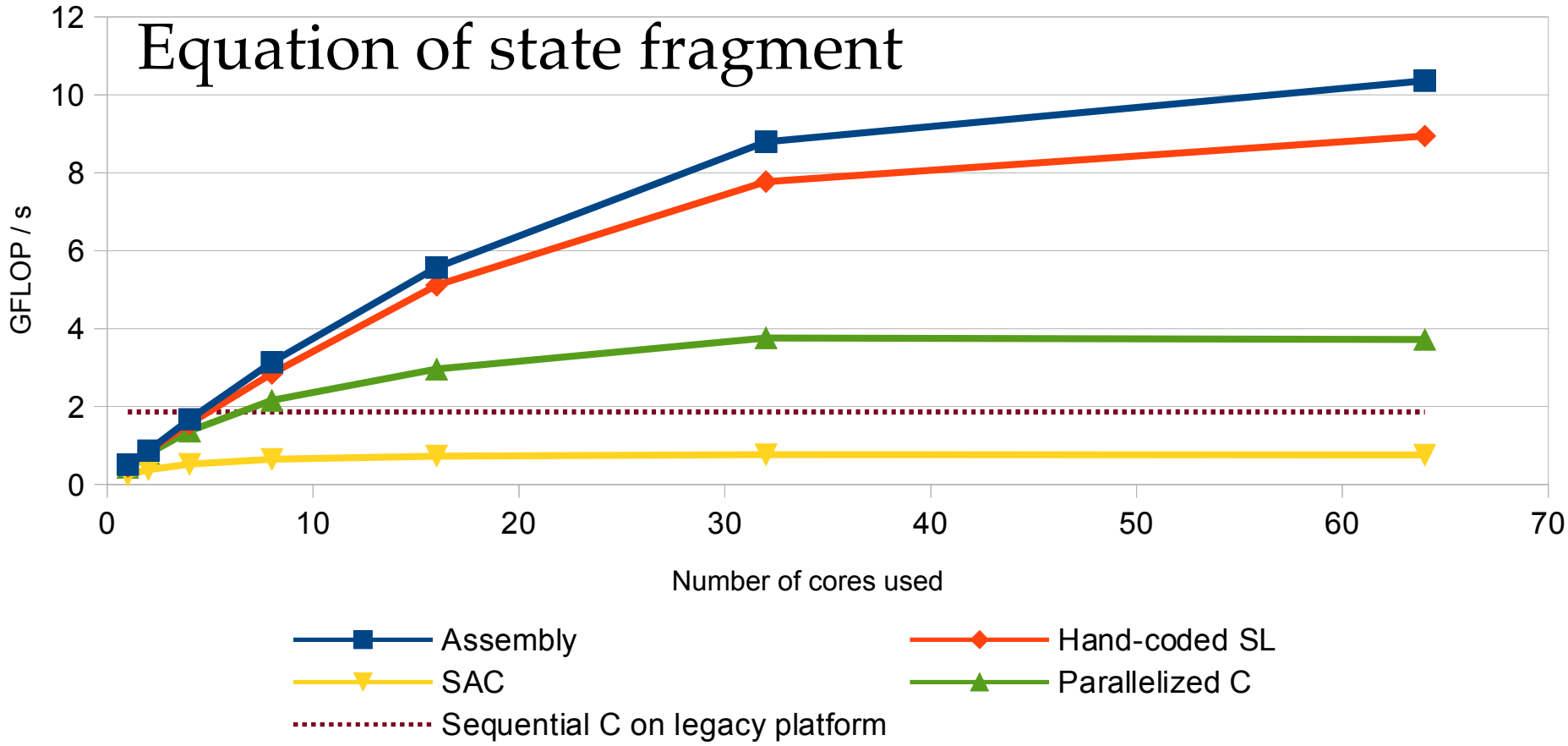
Extra - results



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)