Apple-CORE: SVP and Microgrids

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

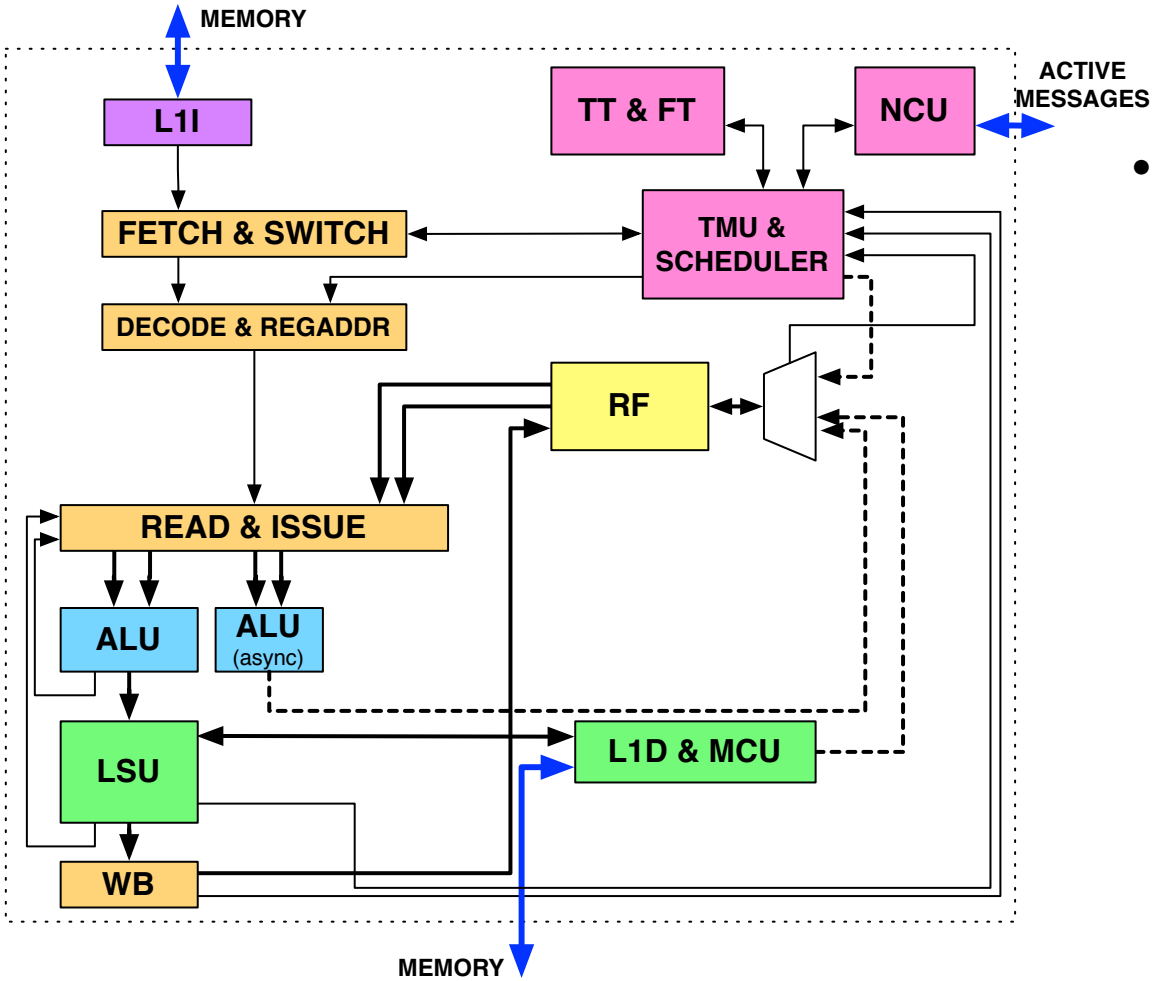
Raphael 'kena' Poss
University of Amsterdam, the Netherlands

DSD 2012
Cesme, Izmir, Turkey
September 6th, 2012

Current general-purpose multi-cores are 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**?

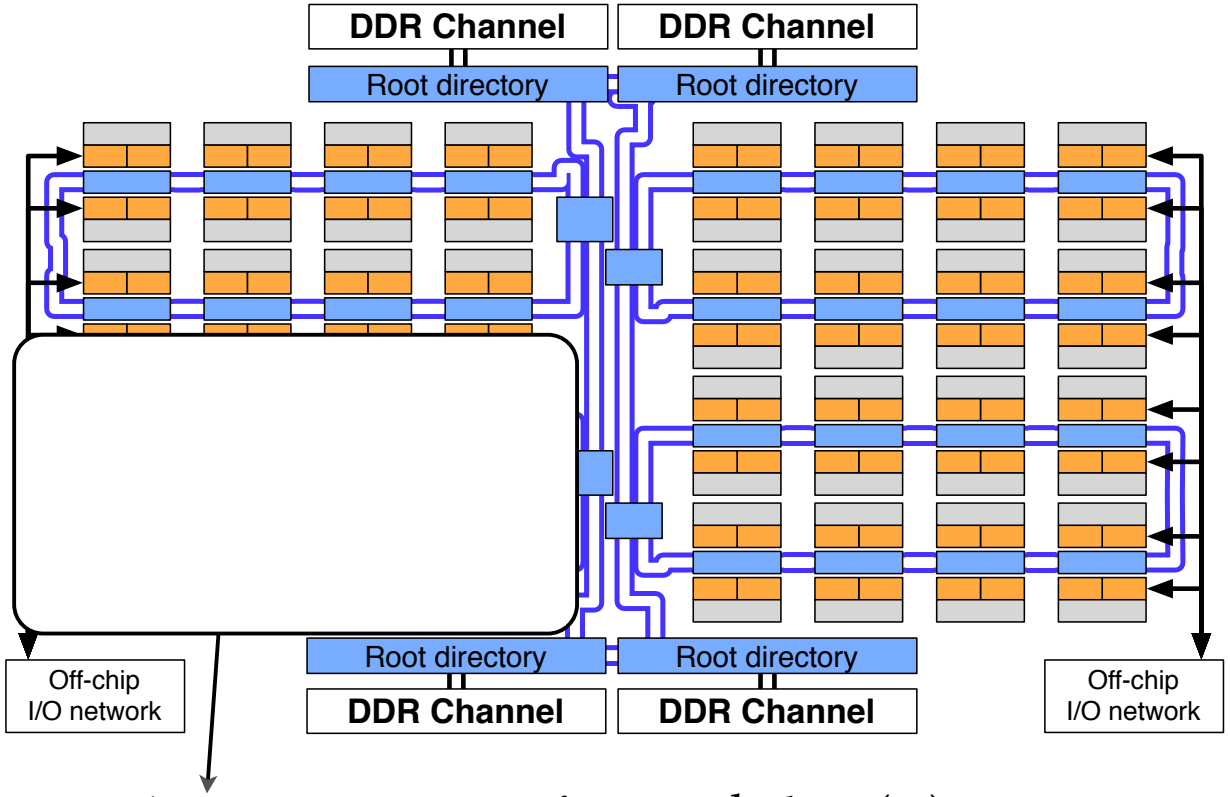
Microgrids of D-RISC cores



- D-RISC cores:
  hardware multithreading + **dynamic dataflow scheduling**
  - **fine-grained threads**: 0-cycle thread switching, <2 cycles creation overhead
  - **ISA instructions and NoC protocol** 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

Example 128-core Microgrid



Approximate size of one Nehalem (i7) core for comparison

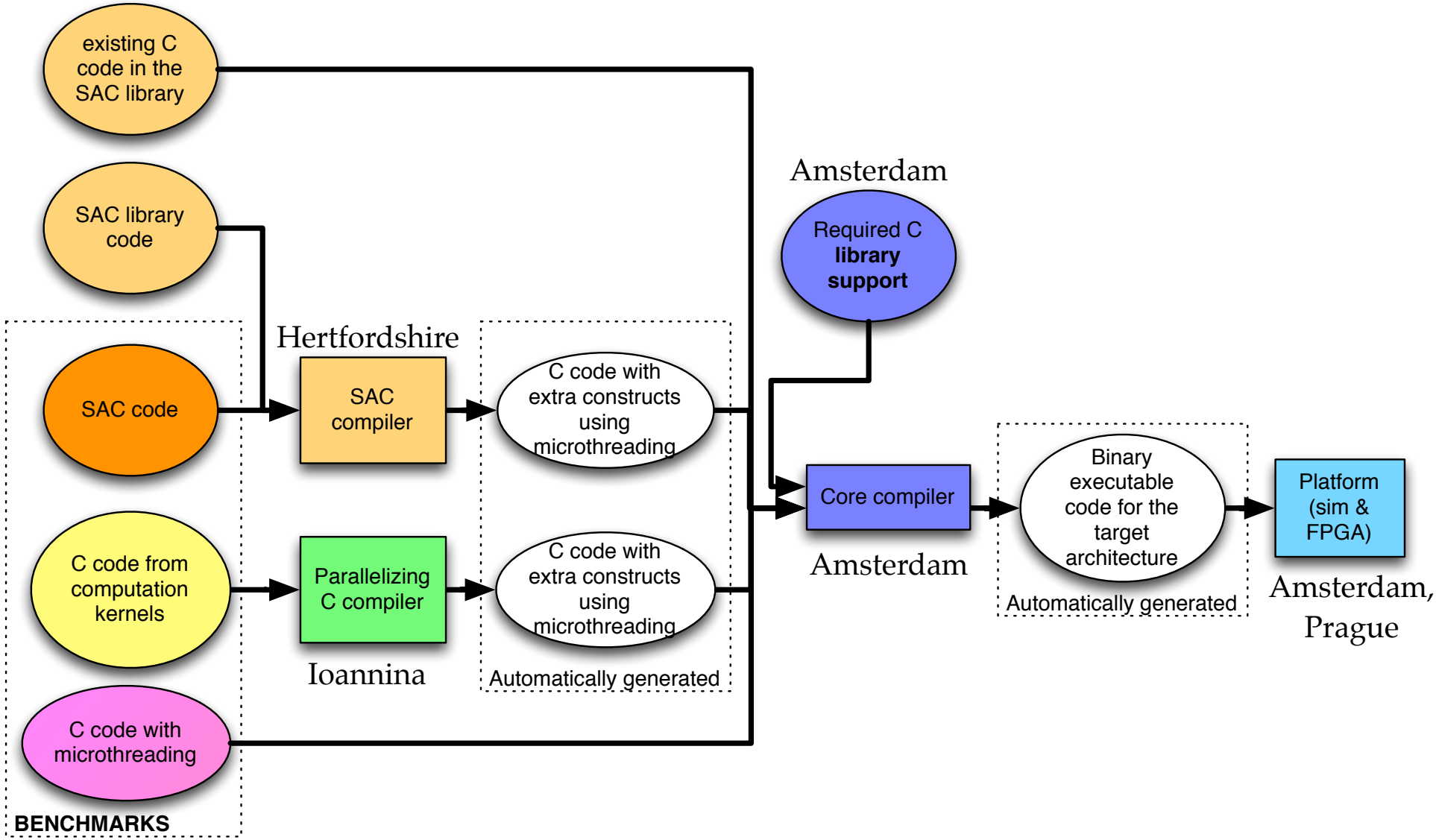
- **32000+ hw threads**
- 5MB **distributed cache**
- **shared MMU = single virtual address space,** protection using **capabilities**
- **Weak cache coherency**
- **no support for global memory atomics –** instead synchronization using point-to-point messaging

Area estimates with CACTI: 100mm2 @ 35nm

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

The Apple-CORE software strategy



The "main" issues uncovered in Apple-CORE

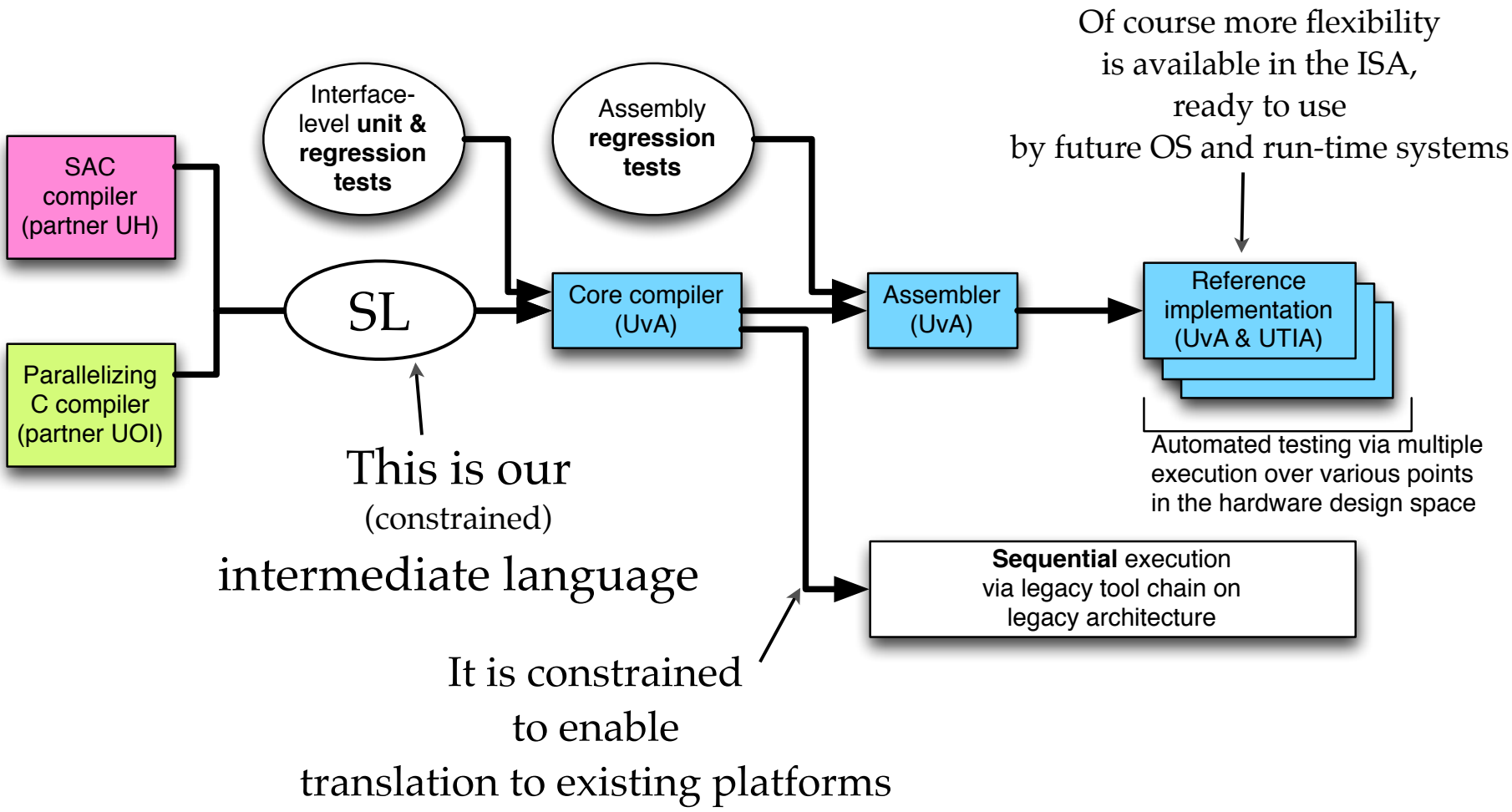
- **Validation**: how to detect detect errors, then compare with existing systems
  – need reference / base lines
- **Resource management**:
  cores, but also **memory** and NoC **channels**
  – how to reduce management overheads

- NB: these issues are general to all many-core processors, but exarcerbated in Apple-CORE

Validation

- Solution:
  1. Choose a **subset of the ISA** that can be emulated in legacy platforms
  2. Design the intermediate language SL to use only this subset to **constrain programs**
  3. Implement **compilation to both** the new platform and legacy systems and perform **comparative testing**
- This subset resembles **fork/join with families** and **forward-only dataflow synchronization**
- It is **deadlock-free**, mostly **deterministic** and **can be serialized** (cf Cilk, Chapel)

Validation


Of course more flexibility
is available in the ISA,
ready to use
by future OS and run-time systems
Interface-level **unit & regression tests**
Assembly **regression tests**
SAC compiler (partner UH)
Parallelizing C compiler (partner UOI)
SL
Core compiler (UvA)
Assembler (UvA)
Reference implementation (UvA & UTIA)
Automated testing via multiple execution over various points in the hardware design space
This is our
(constrained)
intermediate language
Sequential execution via legacy tool chain on legacy architecture
It is constrained
to enable
translation to existing platforms

Resource management

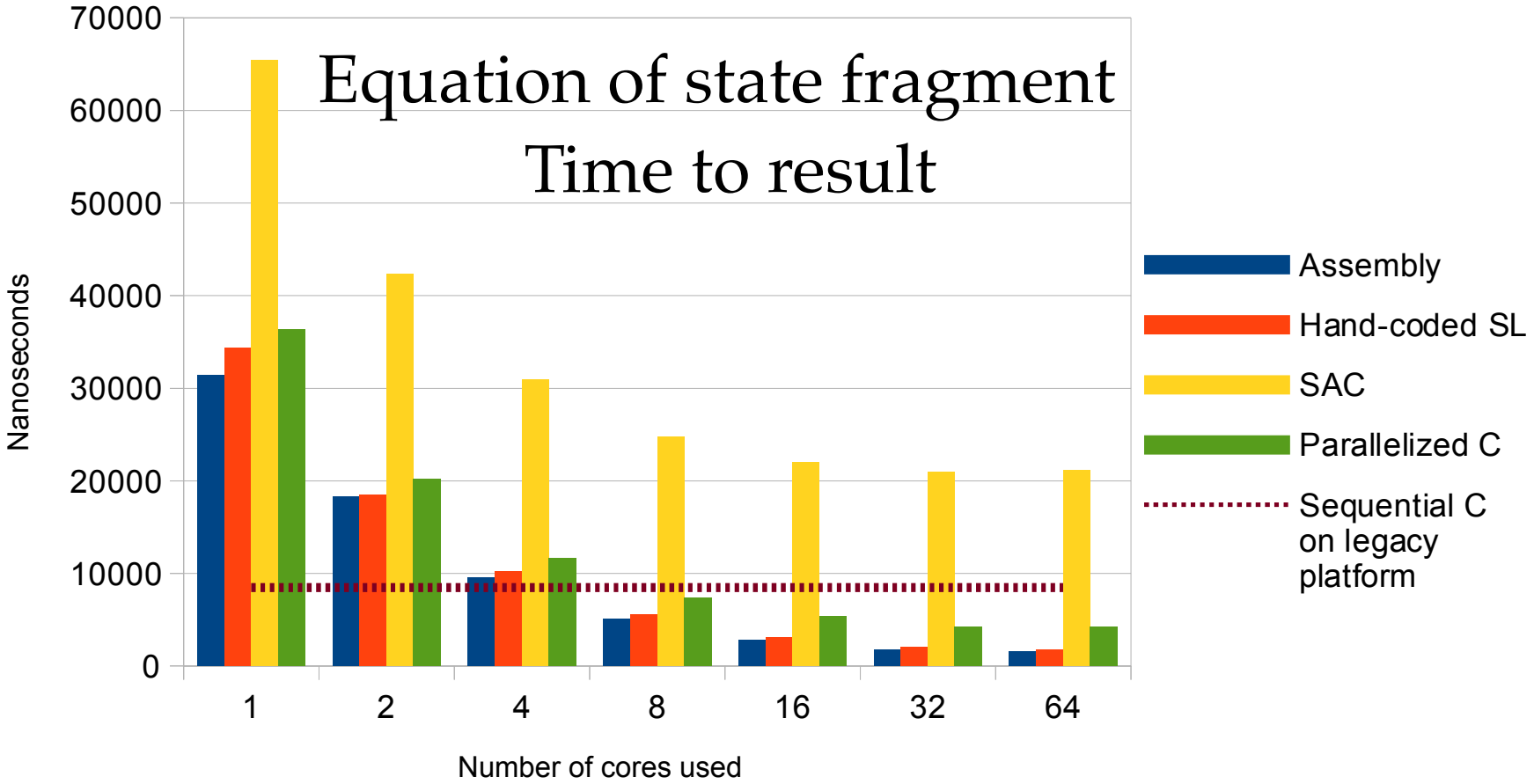
- At the **finest grain**:
  provide TLS to threads created by TMU
  Solution: **pre-allocate** and **partition statically**
- **Concurrency resources:** let programs define more concurrency than available, serialize on demand
- **Algorithms:** distributed memory allocator, garbage collection using reference counting

Resource management

- **Application components**:
  OS allocates and deallocates cores, memory and network links for top-level family entry points
  – this is called **SEP** and is **distributed**
- Either **explicit allocation** in programs

  Or **annotated static requirements**, aggregated at run-time by RTS/OS

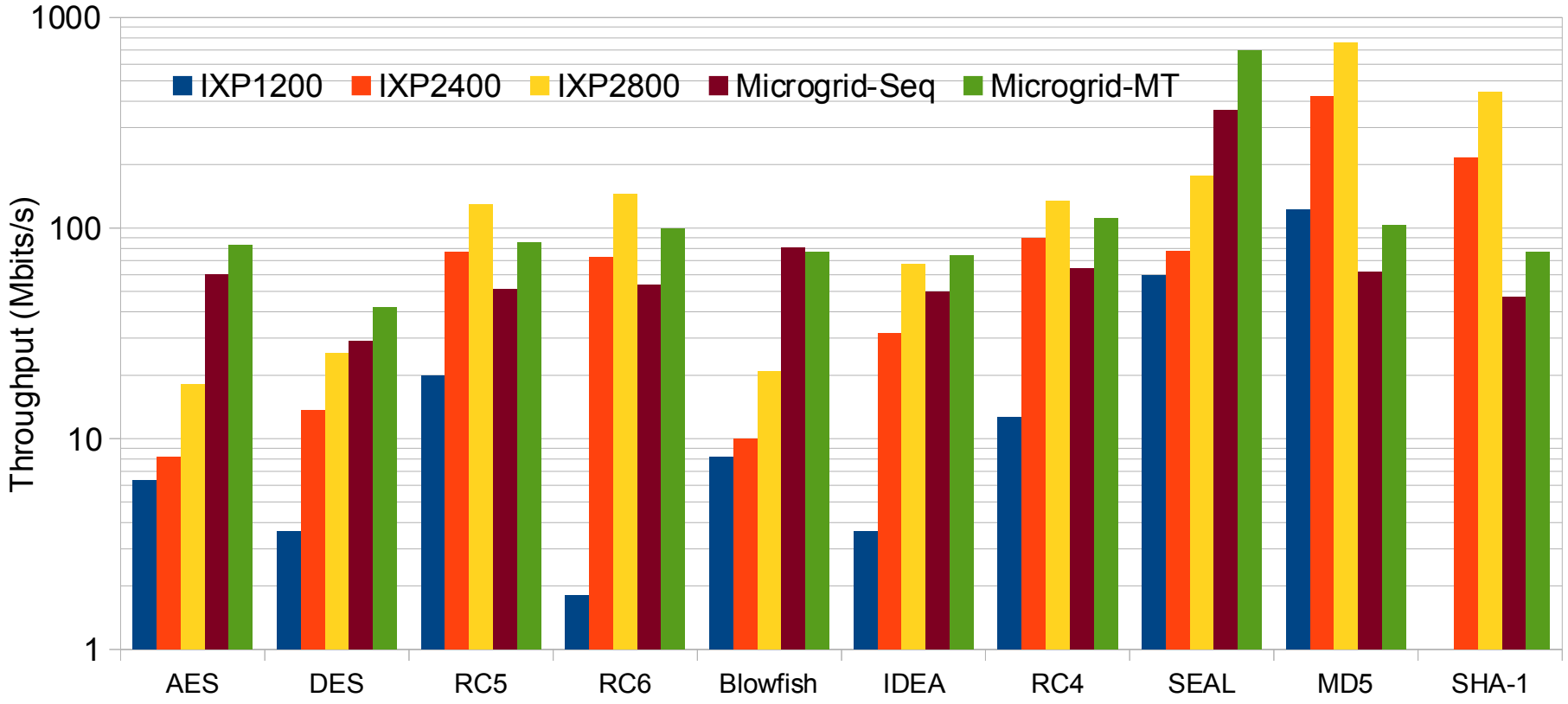
Results: memory-bound kernels



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

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

Results: throughput workloads



Intel IXP = embedded processor specialized for cryptographic workloads

Main results: **Microgrids are general-purpose**, ie not specialized
yet compete on throughput with state-of-the art specialized hardware

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?

Thank you!

- More information:
  - http://www.apple-core.info/
  - http://www.svp-home.org/

SVP 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**