Operating systems in hardware

Scaling from 10 to 1000 cores

Raphael 'kena' Poss
University of Amsterdam, the Netherlands

In cooperation with
Chris Jesshope, Mike Lankamp, Michiel van Tol, and many others
at the CSA group

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**?

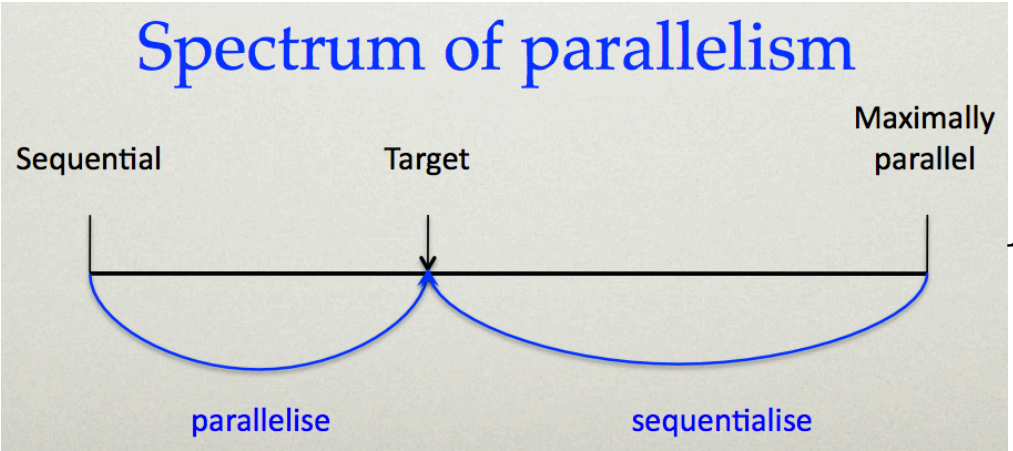
The Apple-CORE approach

- Hardware threading is not new... Denelcor HEP introduced this 30 years ago
- Apple-CORE's contribution is in the *system interface* with a *consistent view on concurrency management* whatever resources are available
  - from a single thread to many cores
- The goal is to *make concurrency the norm* and the processing resource fungible
  - like money it must be divisible and interchangeable

The Apple-CORE approach

- With massive concurrency predicted, the Apple-CORE approach is to:
  - capture maximal concurrency in the binary interface
  - automatically sequentialise it based on resource availability at run time

Past approach



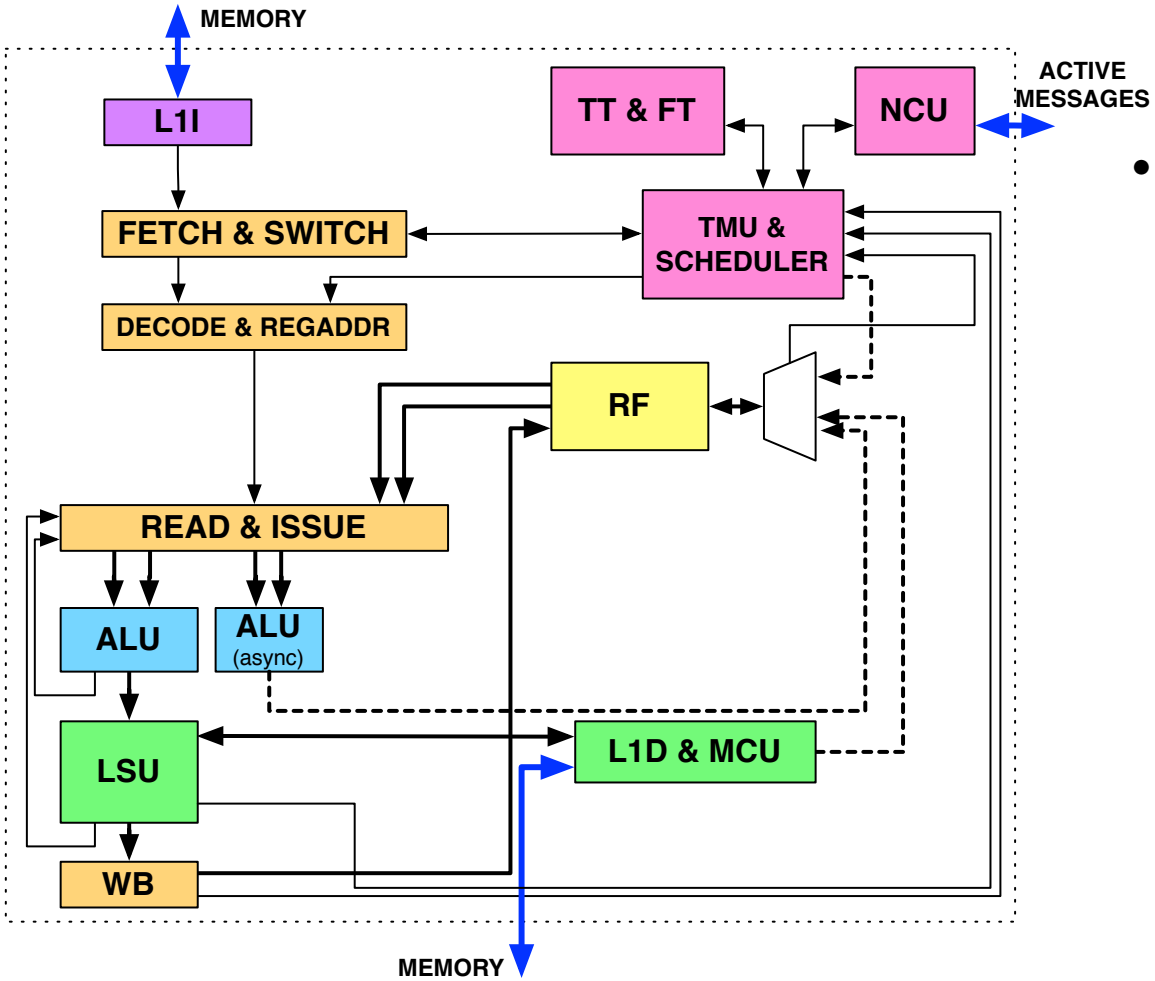
Apple-CORE approach

Key abstractions

- Dataflow synchronisation on instruction execution - *latency tolerance*
- Deterministic programming model based on multi-way fork/sync - *create families of threads*
- Memory consistent at fork and sync events
- Handles on resources - *places*
- Asynchronously execution of *families* at *places*
- Capture sequence where necessary by thread- to-thread dependencies - *shared variables*

The D-RISC core

D-RISC cores

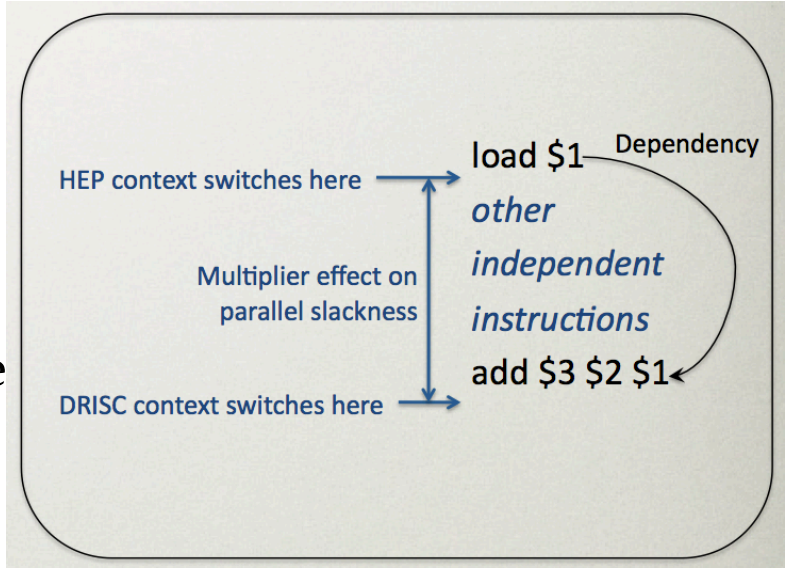


- D-RISC cores:
  hardware multithreading +
  dynamic dataflow scheduling
  - **fine-grained threads**: 0-cycle thread switching, <2 cycles creation overhead
  - **ISA instructions and NoC protocol** for thread management
  - dedicated hardware processes for **bulk creation and synchronization**
  -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

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

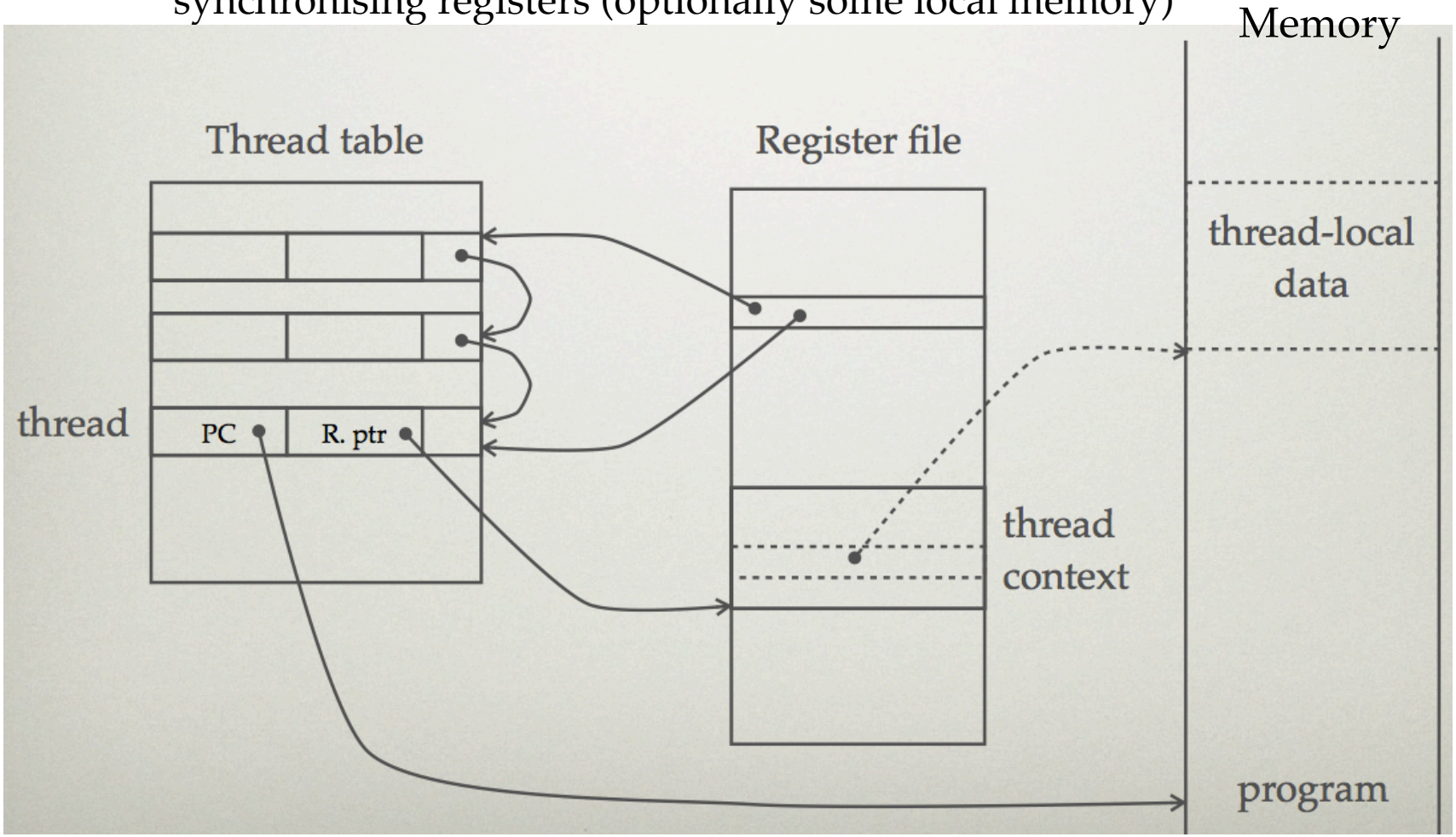
Dataflow scheduling

- Dataflow is a more effective way to tolerate latency in instruction execution - e.g. memory accesses take 100s or even 1000s of cycles
- Prior approaches e.g. HEP/ Niagara context switch on the hazard-inducing instruction
- DRISC synchronises on the result of the hazard, i.e. captures the dependency - *dataflow scheduling*
- This requires a synchronising register file (i-structures) and an efficient mechanism to store and manage thread continuations



D-RISC CONTINUATIONS

- Continuation comprise a PC and a context comprising from 2 to 32 synchronising registers (optionally some local memory)



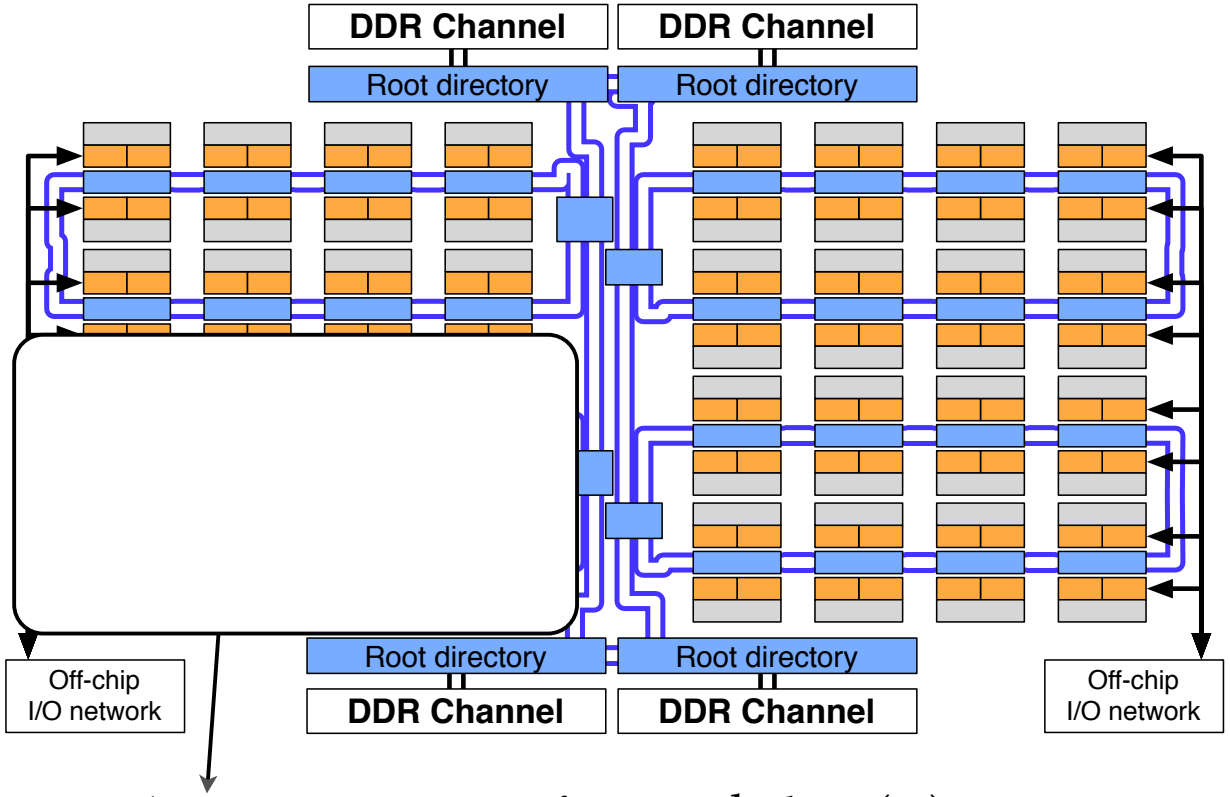
Asynchrony in D-RISC cores

- *Intra-thread* - i.e. an instruction that is issued and synchronised within the same thread:
  - Memory references
  - FPU operations (NB: shared FPU)
  - Functions implemented as logic
  - Create a family and Sync on termination of a family
- *Inter-thread* - read/write to a remote register file
  - This asynchronously activates a waiting thread and is one of two mechanism used for system signalling - the other is asynchronously creating a family of thread(s)

Achievements - D-RISC cores

- *Single DRISC core* - FPGA implementation of single core (SPARC ISA)
- *Many-core Microgrid* (Alpha and SPARC ISAs)
- Low-level *C-based compiler, operating system and run-time environment* for both platforms

Example 128-core Microgrid



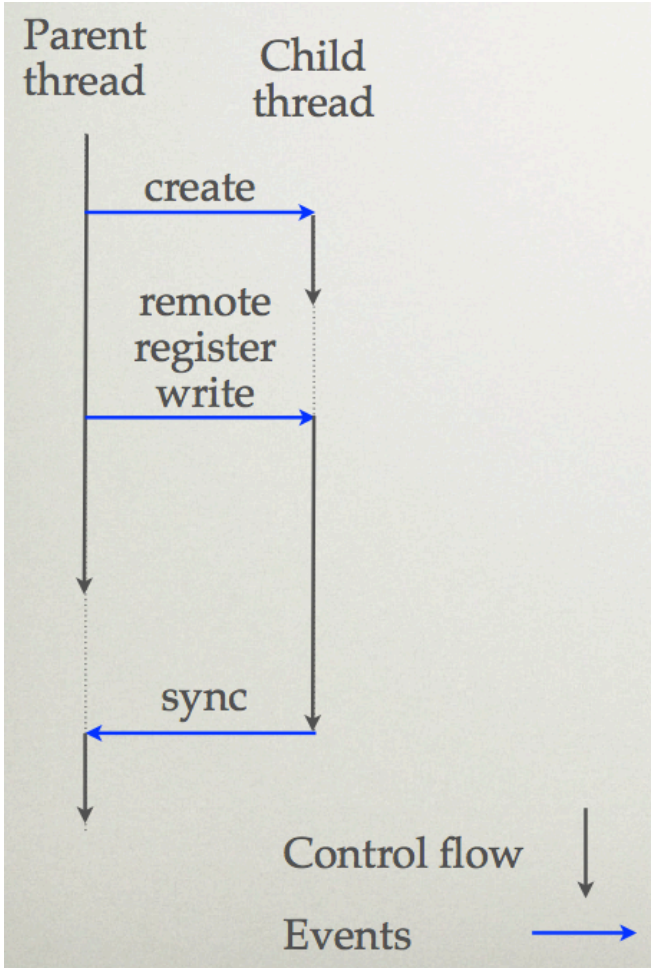
Approximate size of one Nehalem (i7) core for comparison

- **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

Microthreaded programming

Functions as threads

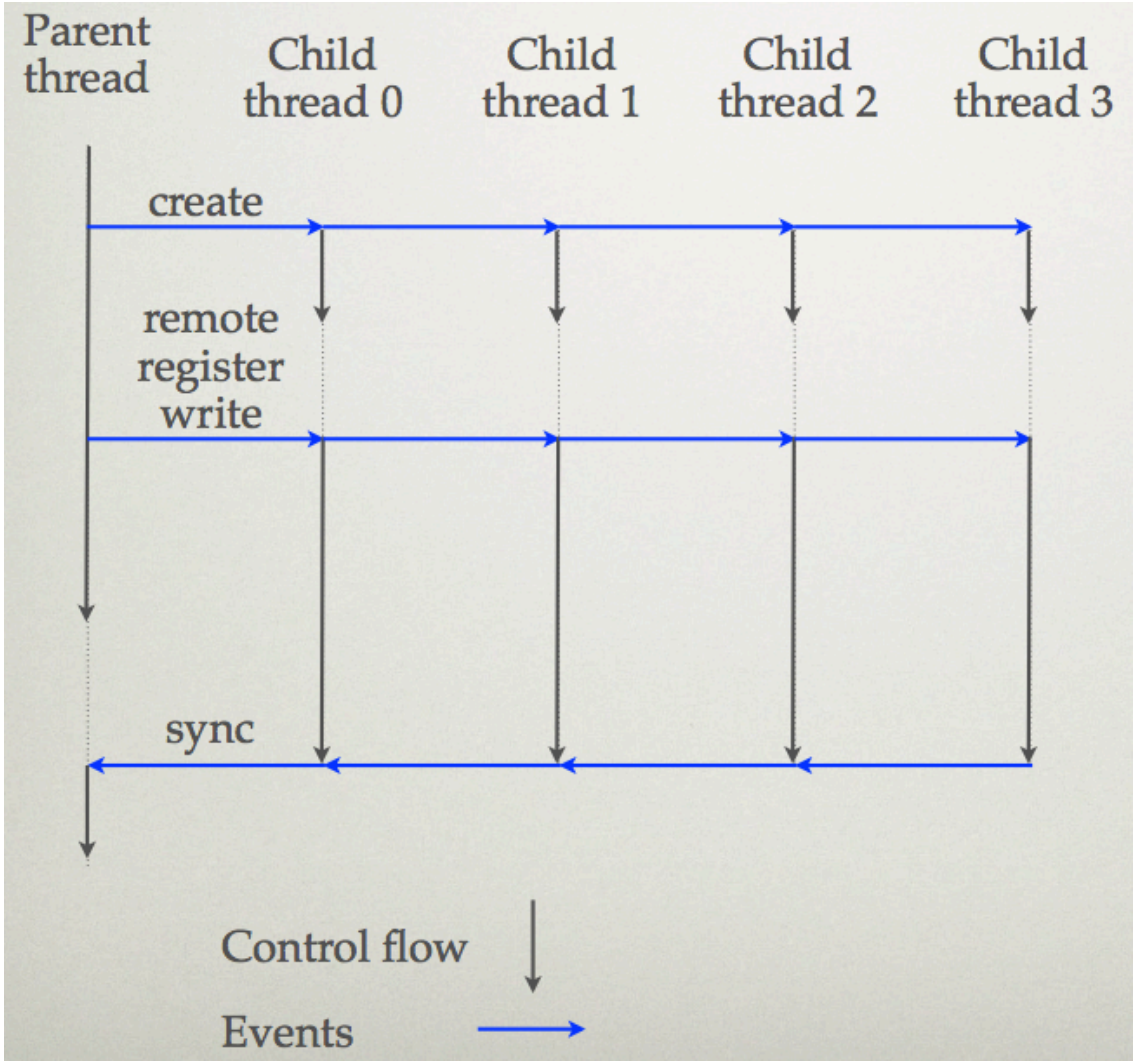


- Events are **create/sync** and **remote register read/writes**
  - register writes are synchronising
  - asynchronous parameter passing
  - create executed early and parent and child execute concurrently

```
foo(A)
{...
...A...
}...
```

```
...
create() foo(P)
... P:=... ...
sync
```

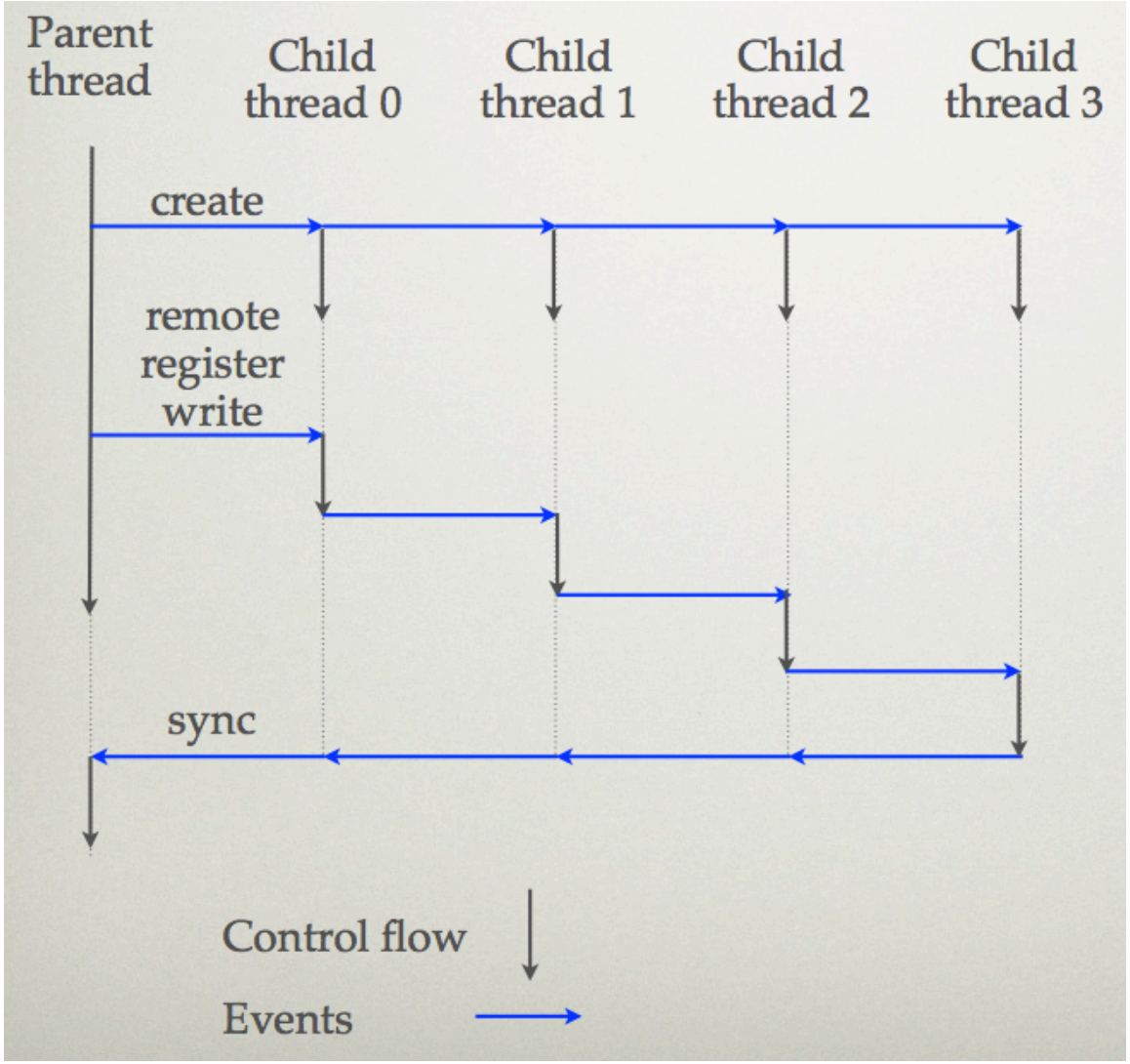
Loops as threads




```
foo(A)
{...
...A...
}...

...
create(i=0..3)
    foo(P)
... P:=... ...
sync
```

Sequence as threads




```
foo(A)
{...
 A := ...A...
}...

...
create(i=0..3)
    foo(P)
... P:=... ...
sync
```

Why capture sequence as threads?

- Still need local concurrency for latency tolerance.
  e.g. *naive inner product*
  - sum:=sum+A[i]*B[i]
  - compiles to a six instruction thread in Alpha
  - `A+i, B+i, load, load, fmul` are all independent
  - only one instruction: `fadd $S1 $D1 $L1` is constrained to be executed sequentially
- No speedup, but *independent instructions can be scheduled concurrently* on a single core
  - This allows multiple concurrent memory loads and floating point multiplies to be executed concurrently on a single core

Programming environment

- C with extensions - μTC for documentation, SL in implementations
- New *language primitives* for concurrency creation, synchronization
- A *thread program* is like a C function with special constructs for dataflow channels
- *No model assymmetry* (like with CUDA/OpenCL): any thread program can use any library service
- To be *targeted by higher-level compilers*

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

Operating systems for microgrids

Evolution or revolution?

- Looks like a *revolution*:
  - **Can't (shouldn't) reuse existing OS** code as-is
  - **Can't reuse existing low-level techniques**
    based on preemption and software schedulers
    eg. signals, interrupt handlers, "callbacks"
  - Must use *ISA concurrency in code generation* to exploit; requires *language extensions* and shakes *compiler assumptions*
- Can we really afford this?

An evolution, really (1)

- Low-level machine code generation:
  - *Lift loop bodies as separate functions*
    - reuses techniques from GPU/accelerator world
  - *A thread is really a virtual processor*
    – threading is well-know in compilers already
  - *Higher-level compilers* can generate threaded low-level code from “productivity” languages

- Really a **convergence** of mature technology

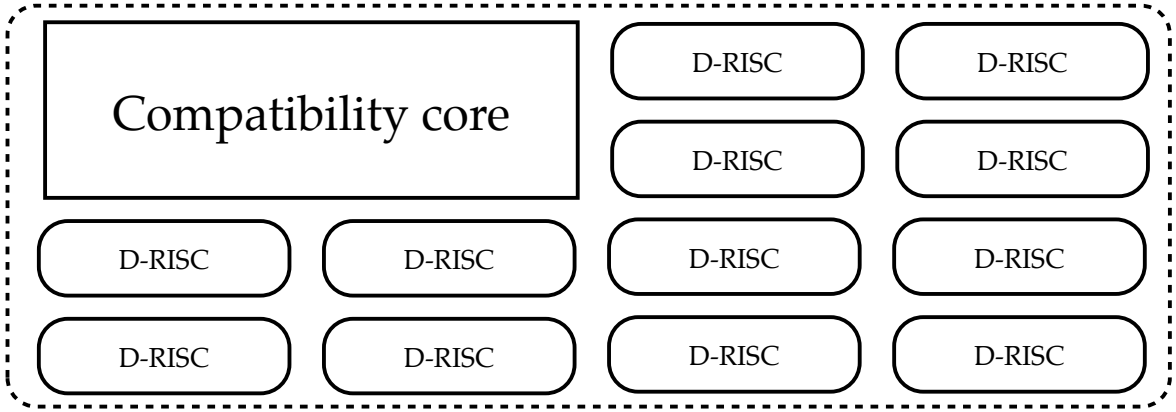
An evolution, really (2)

- Managing asynchrony of "external" events
  – I/O, traps, remote "syscalls"
  - *An event handler is really a thread*
    – reuse as is, just entry/exit is different
  - Requires *mutual exclusion of shared state*
    – already accepted in OS/library design

- The benefit of extra bandwidth and lower latency will justify the req. adaptation, if any

An evolution, really (3)

- Sometimes ***legacy OS and library code*** cannot be adapted – typically device drivers, proprietary interfaces



- Solution: integrate a "compatibility" core on the same chip using ***same NoC protocol for concurrency management***, then ***delegate syscalls*** behind library entry points
- With same ISA and shared memory ***APIs can be kept as-is***
- ***The "accelerator pattern", transposed!*** – similar to Cray XMT, on chip

Foreground produced

- Technology:
  - various *emulators/simulators* for a many-core chip with hardware concurrency management (Microgrid)
  - MGOS: *OS and library components* to drive the hardware architecture, including *resource allocators* and *API bridges*
  - *compilation tools* to/from the SVP intermediate language
  - *software run-time systems* for commodity multi-cores using SVP semantics
  - Tests and benchmarks to validate and evaluate fine-grained concurrency management
- + Accompanying documentation & know-how

OS Summary

- A true ***perspective shift*** for the basic OS/compiler abstractions:
  - from *sequence* to *concurrency*
  - from *loops* to *microthreads*
  - from *function calls* to *family creation*
  - new focus on ***placement*** and locality
- Revolution in hardware, yet only an evolution in software
- Middle ground: a common set of primitives in ISA
  = ***a concurrency management protocol*** on chip
- This is generalized from D-RISC towards portable SVP

The "main" issues uncovered in Apple-CORE

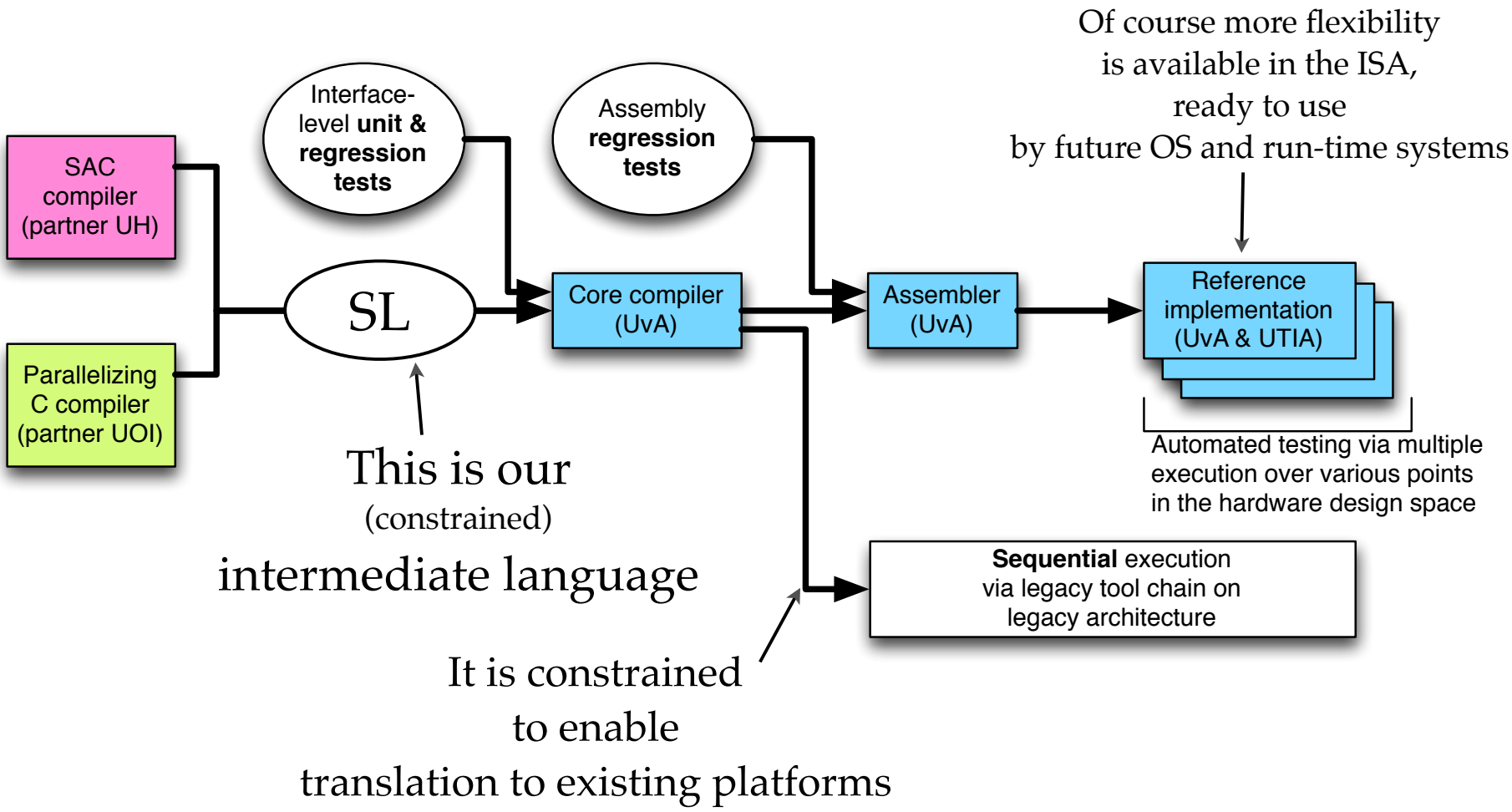
- **Validation**: how to detect detect errors, then compare with existing systems
  – need reference / base lines
- **Resource management**:
  cores, but also **memory** and NoC **channels**
  – how to reduce management overheads

- 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*, *deterministic* if race-free 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

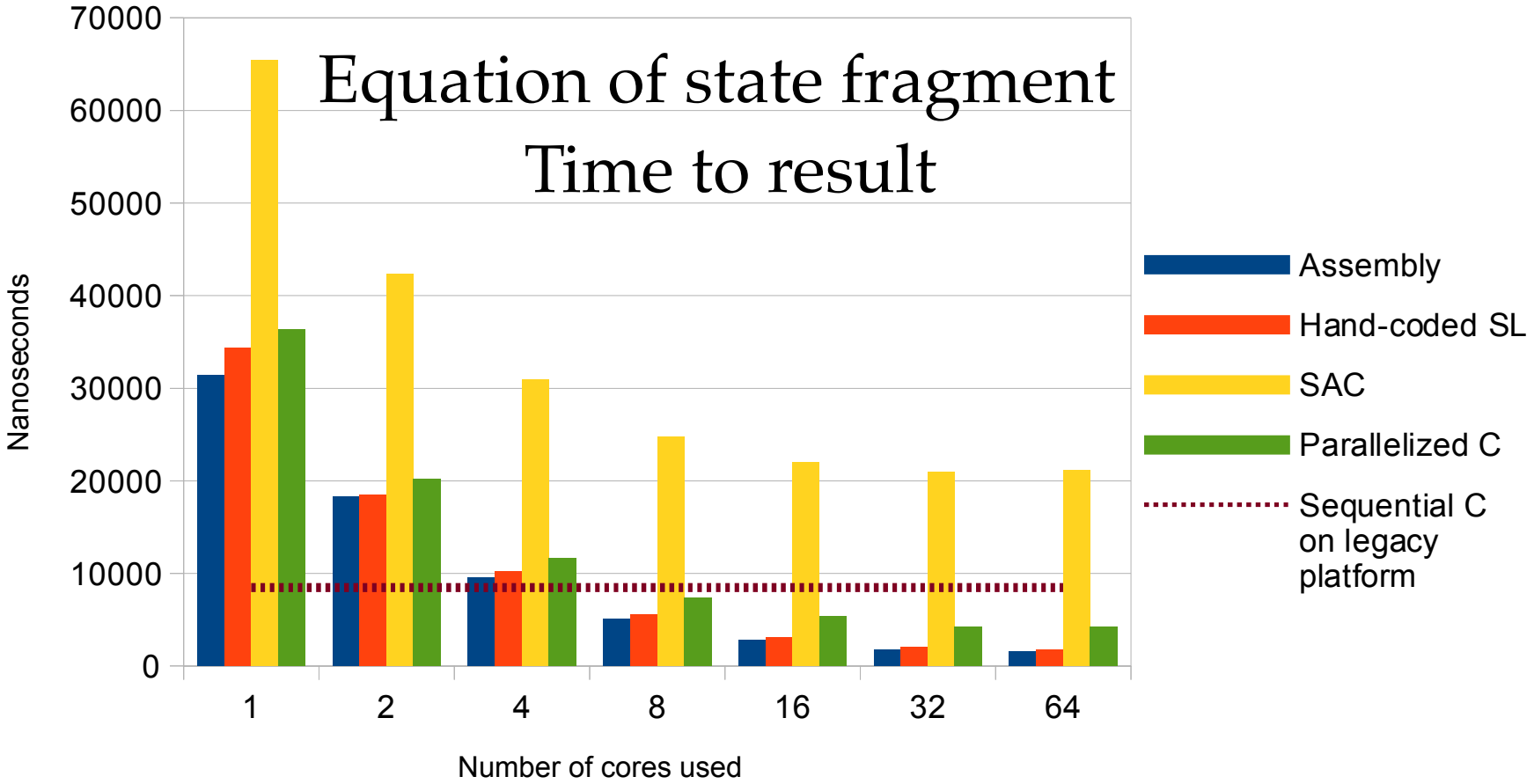
- At the finest grain:
  provide TLS to threads created by TMU
  Solution: *pre-allocate* and *partition statically*
- *Concurrency resources*: let programs define more concurrency than available, serialize on demand
- *Algorithms*: distributed memory allocator, garbage collection using reference counting

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

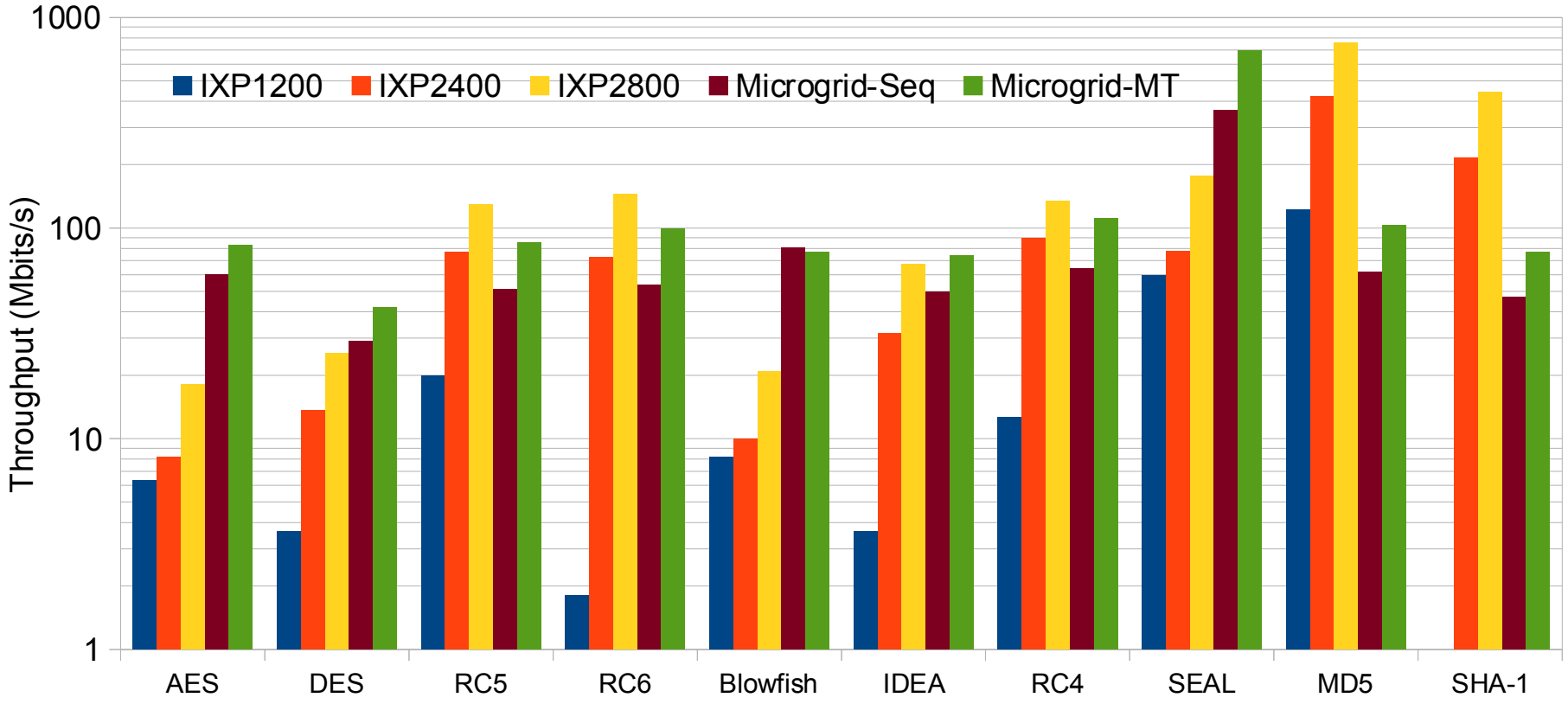
Results: memory-bound kernels



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?

- *Internal* issues: memory consistency, scalable cache protocols, ISA semantics, etc.
- *External* issues from outside architecture: platform virtualization, space scheduling, I/O device drivers
- *Fundamental* issues: concurrent complexity theory

Thank you!

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

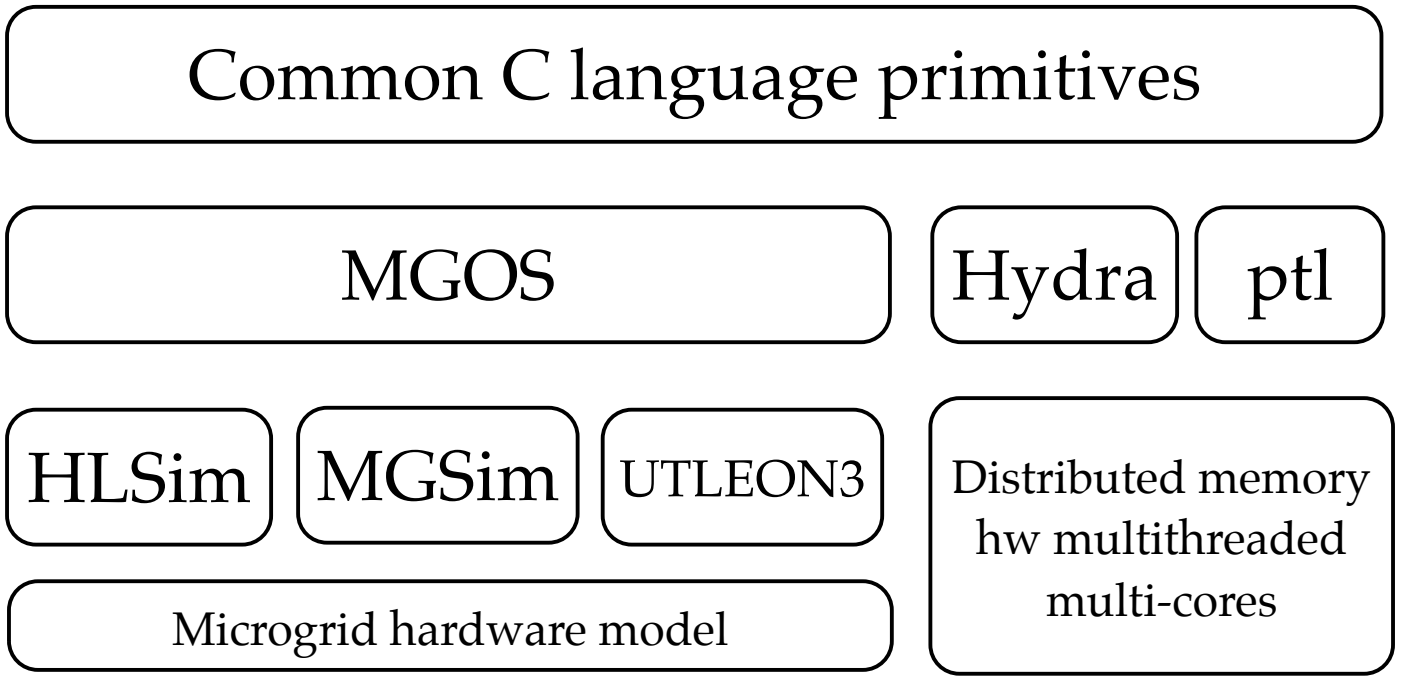
SVP Concurrency Management Protocol

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

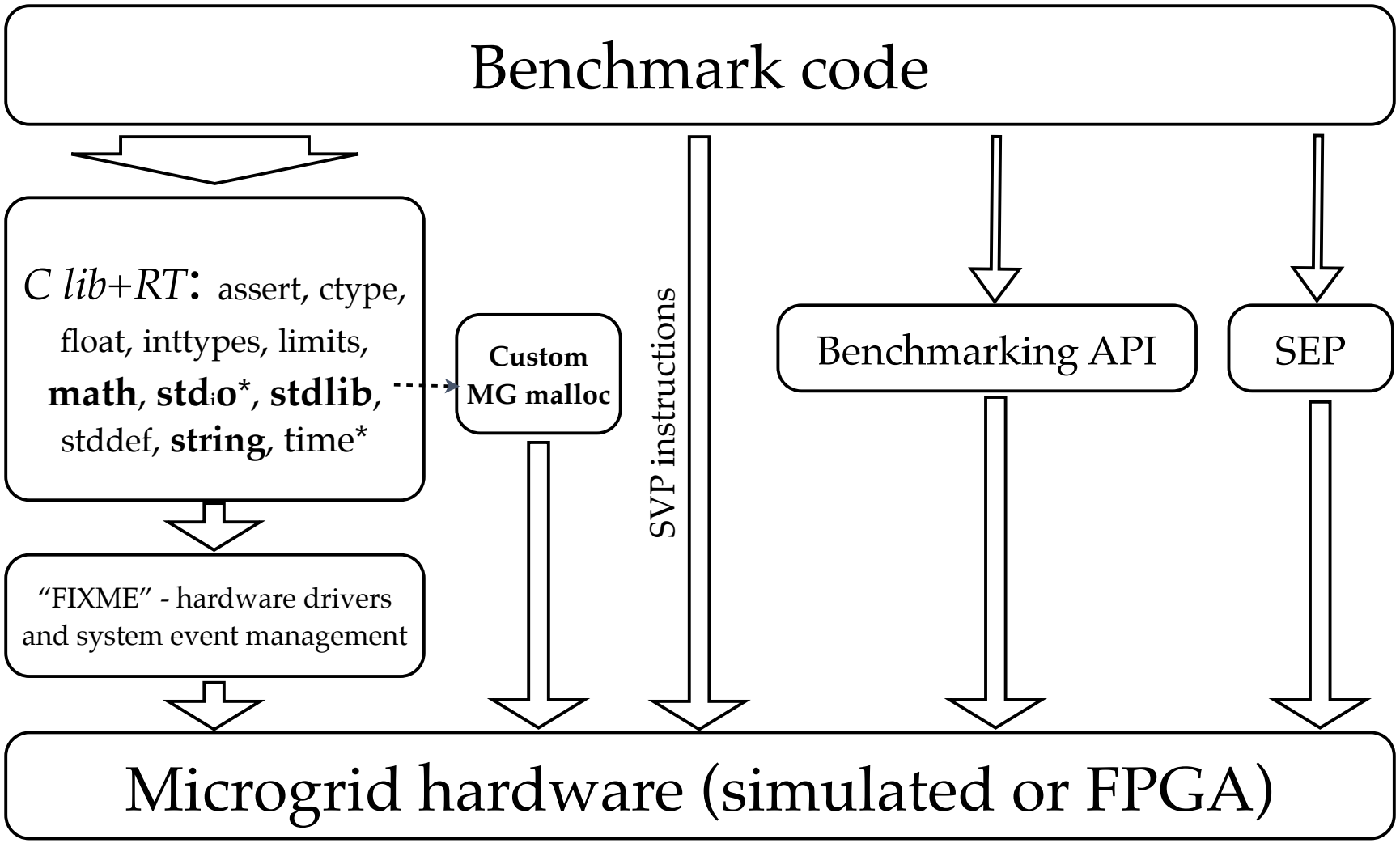
Extra - A perspective shift

	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**

Foreground produced

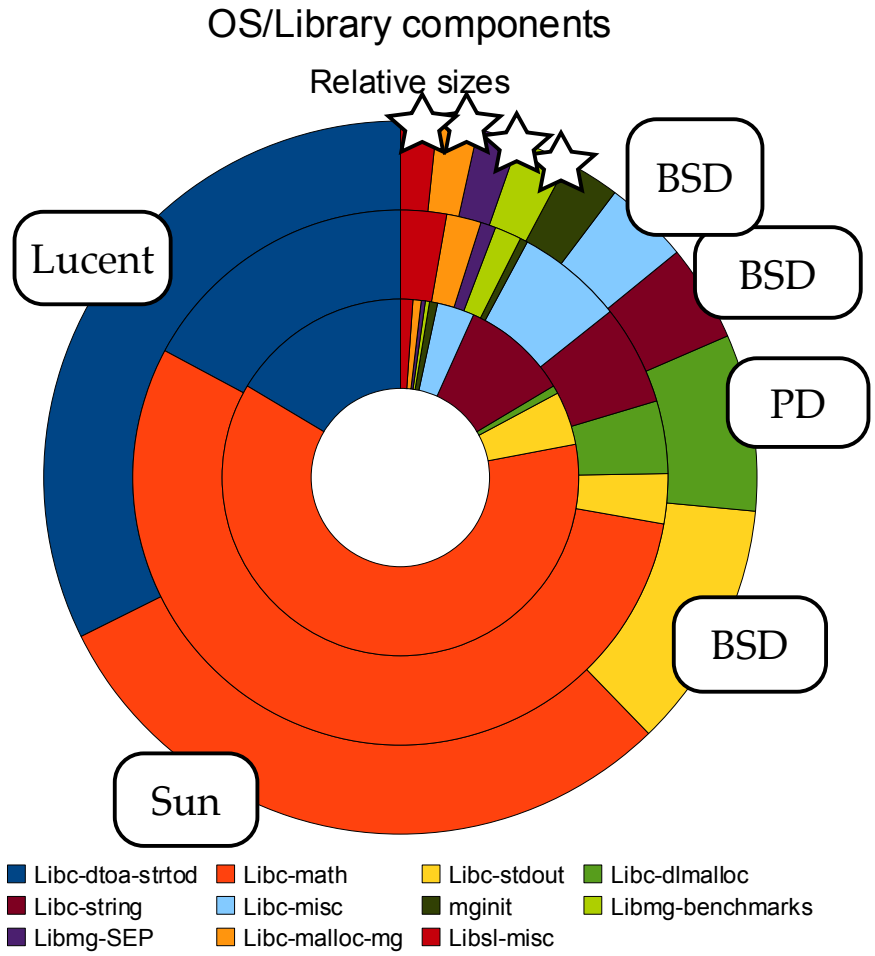


MGOS: Software interfaces



Inside the MGOS

Component	Objects	Functions	Code words
Libc-dtoa-strtod	44	74	14528
Libc-math	165	238	13424
Libc-stdout	13	13	5040
Libc-dlmalloc	2	19	3600
Libc-string	26	26	1968
Libc-misc	9	28	1728
mginit	2	2	1120
Libmg-benchmarks	1	7	1072
Libmg-SEP	1	4	848
Libc-malloc-mg	2	9	832
Libsl-misc	3	12	736



Many copyright holders...

... but only one import tree: FreeBSD

And comparatively little MG-specific code!