Network-Driven Processor (NDP): Energy-Aware Embedded Architectures & Execution Models

Princeton University

Collaboration with Stefanos Kaxiras at U. Patras

Patras & Princeton: Past Collaborations

- Cache Decay
- Timekeeping Prefetch
- Tag-Correlating Prefetch

- Natural to consider ways of continuing collaboration…

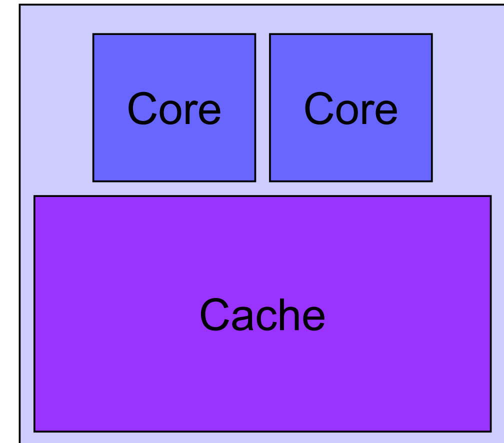
Motivation

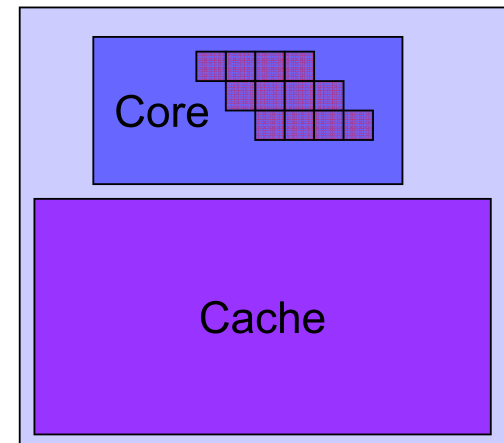
- Increasing trend towards concurrency: SoCs, CMPs, multi-core architectures
    - IBM Power5, Power6, Sun Niagara, ARM MPCore
- Why on-chip parallelism?
    - Moore's Law has given us enough transistors
    - Replicating cores mitigates design complexity
    - Easy tricks to boost single-core performance are running out
    - Often (embedded systems) it's the natural approach

Key challenge: How to map workloads/applications to these chips to maximize performance and yet also abide by power/thermal limits?

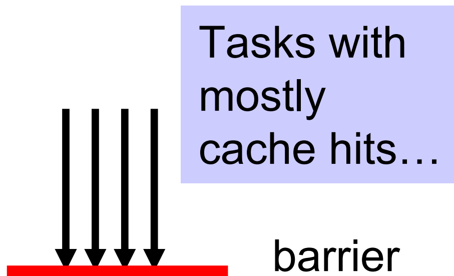
Reducing things to a previously-unsolved problem….. Parallelism!

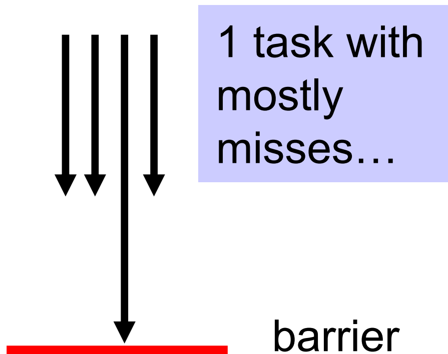
- Parallel workloads have always been hard
  - Managing synchronization
  - Optimizing inter-process communication
  - Load balancing
  - Discovering sufficient parallelism at all…
- Two main approaches:
  - Very coarse-grained parallelism
  - Very fine-grained (ILP) parallelism





Problem: Handling runtime variability





- Current solutions for handling runtime variability incur high overhead
- Example: Memory behavior and data set variability:
  - Cache prefetching is not a silver bullet
  - Dynamic compilation
    - severe overheads: tens of milliseconds
  - Lightweight (user-level) threads packages to hide latencies
    - Software overheads tens of microseconds
      - pipeline drain and reg context switch
  - OS-level multithreading/multiprogramming
    - coarsely interleaves threads and programs
    - Overhead: 100s of microseconds -> 1 ms
- Current trend: many many sources of runtime variability, not just memory!

NDP in a nutshell…

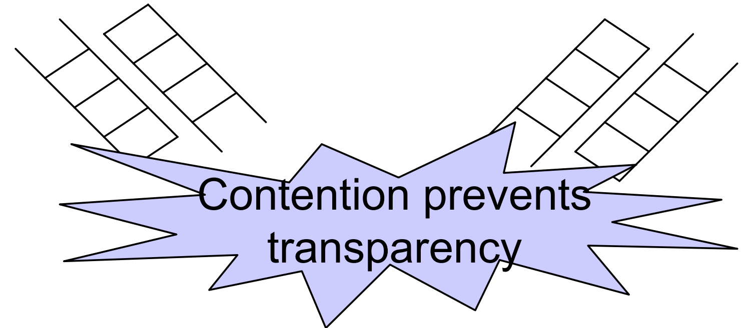
- Hardware support for:
  - Adaptive Parallelism modulation
  - Task placement
  - Load balancing
  - Energy management
- Two key challenges:
  1. Low-overhead mechanisms supporting above features
     - Key insight: A CMP's interconnect fabric has excellent access to the info needed!
     - Engineered specifically to support dynamic management of parallelism and power
     - Track communicate rates and CPU requirements of different threads
  2. Stable, distributed control policies
     - Spawn threads such that related threads are co-located
     - Schedule or migrate competing threads
     - Manage energy and temperature based on same usage stats

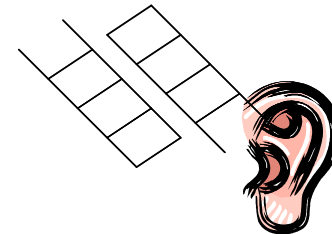
Challenge #1: How can we minimize run-time overhead of application partitioning?

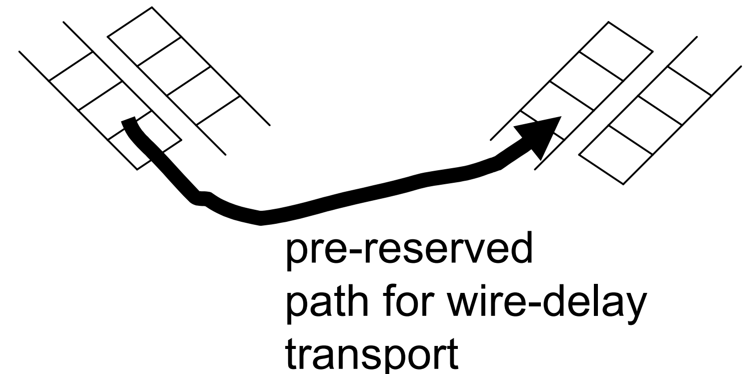
- Key phases of dynamic partitioning
  1. Are we balanced?
  2. If not, where shall we place the execution?
  3. Can we quickly launch execution?
     - Grab instructions
     - Grab data
     - Context-switch

Network-driven architecture lowers run-time overhead of application partitioning

- Are we balanced?
  - Overhead: Communications between producers/consumers
  - 0-overhead solution: Queue transparency through network reservation
- Where to place?
  - Overhead: Multiple communications between tiles
  - 0-overhead solution: Snooping
- Launch execution?
  - Overhead: Communications of data, instructions, context switching
  - Minimal-overhead solution:
    - Reservation permits wire-delay data, instruction transport
    - Message-driven execution permits fast triggering of execution
    - Multiple register contexts enables fast thread context switching







pre-reserved path for wire-delay transport

Proposed hardware tile architecture

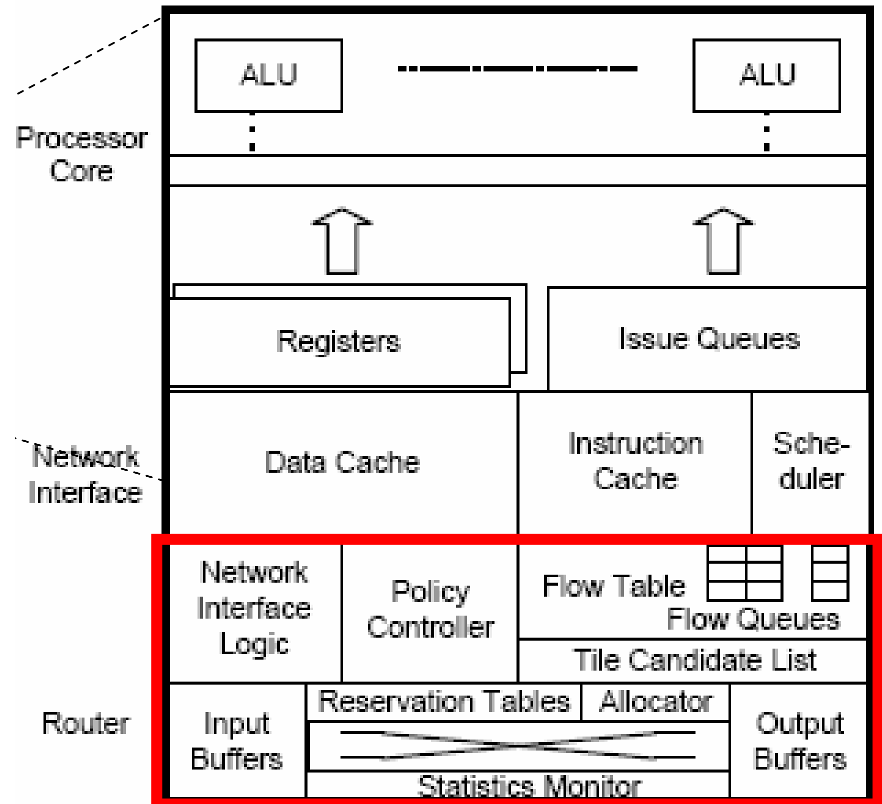
- Flow Table – Keeps track of flows running on this tile and locations of their producers and consumers

- Tile candidate list – Load statistics of tiles maintained by the router

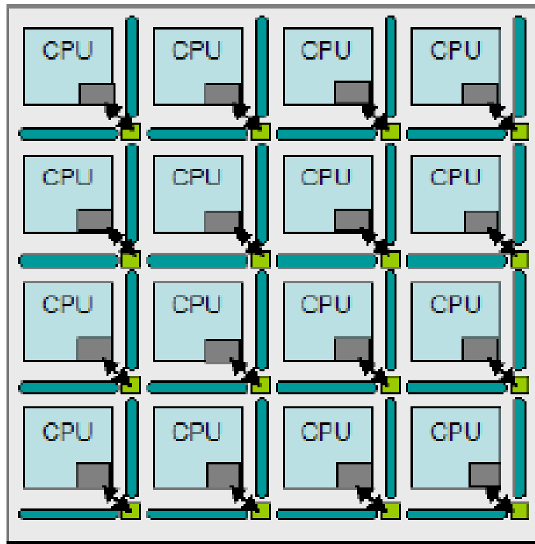
- Policy controller – Logic/brains behind when and where to spawn flows

- Scheduler – Picks and launches local flows onto core pipeline

- NI logic and Router – Reserves circuits to maintain transparency; Statistics snooping; Minimal-overhead communications

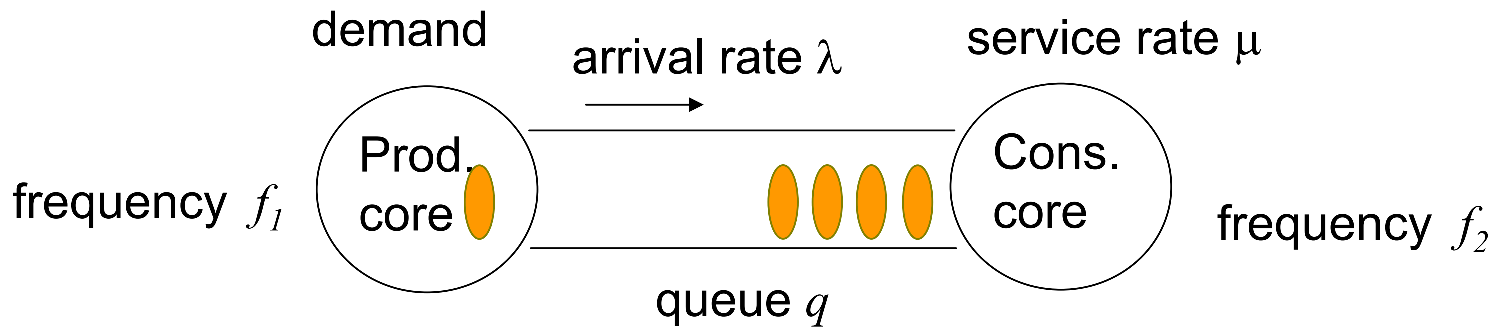


Challenge #2: Energy and Speed-balancing on CMPs



- Not all cores are useful at full-speed at all times…
  - Limited parallelism
  - Memory or I/O stalls
- And may need to adjust to thermal or energy emergencies…
- Via a CMP's inter-core networks, can see data communication relationships
- This work: Dynamically adapt power & V/f settings according to data & CPU usage

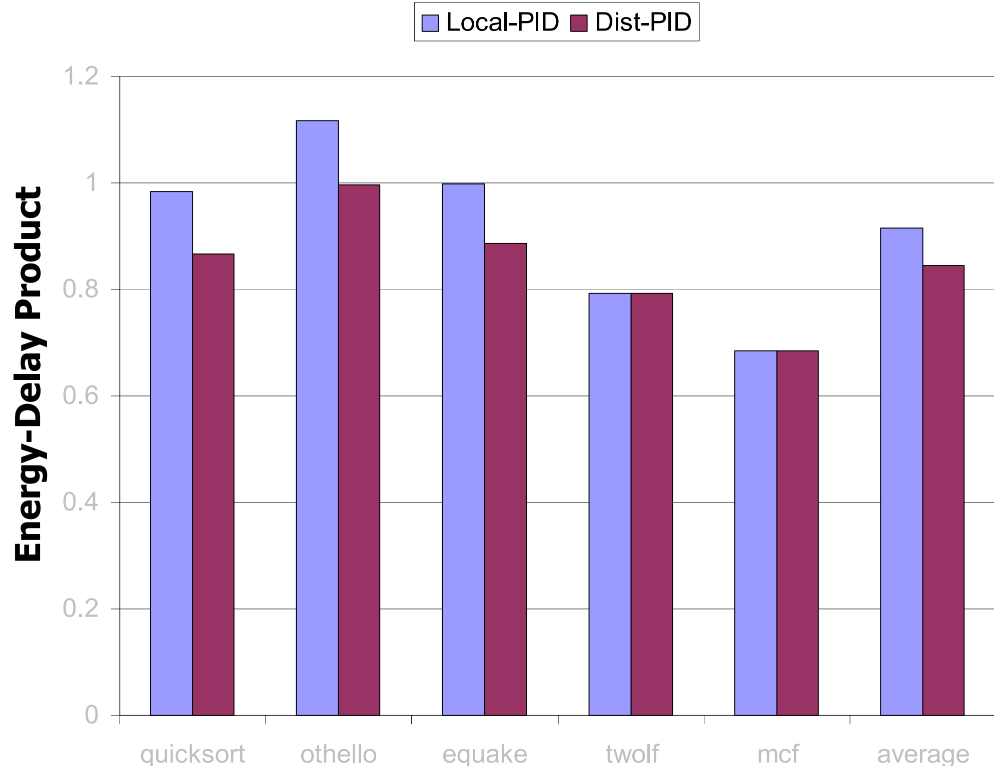
DVFS using Producer-Consumer Cores



- Adjust rates to give "just enough" performance
  - Identify producer-consumer relationships
  - Speed balance based on data "pileups" in between them

Energy-Delay Product: Improvement over a local approach

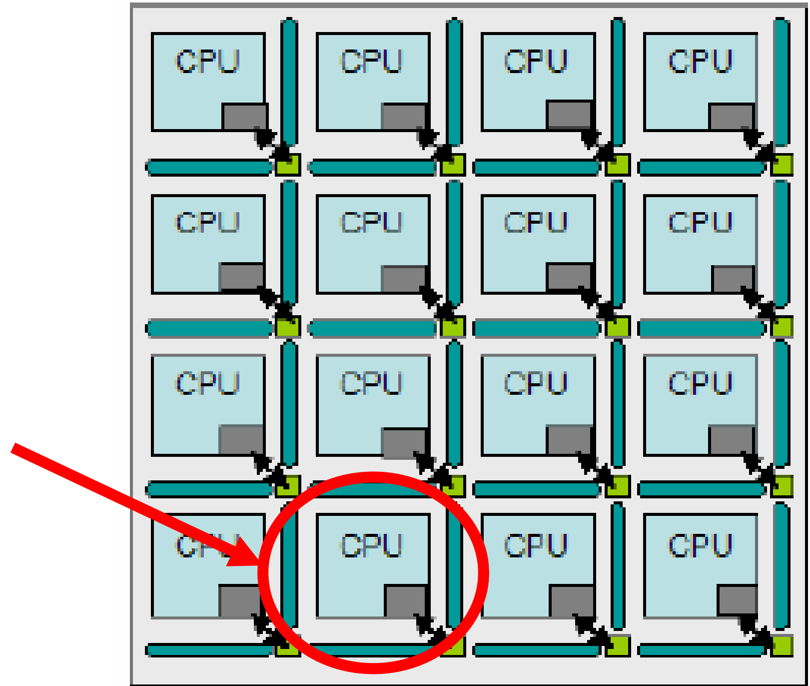
- Quicksort: Fast moving, high thread pressure
- Othello: Slow moving, bursty
- 183.equake: Statically balanced, steady
- 181.mcf: Bimodal
- 300.twolf: Small but significant and easy to identify opportunities



Dist-PID equal or better energy-delay product than Local-PID for all benchmarks

NDP: Beyond homogeneous CMPs…

- No need for CPUs to be identical
  - Vary speed, pipeline to allow richer perf/energy tradeoffs
- Heterogeneity: can replace CPU blocks with ASIC, FPGA, vector units or other specialized hardware
- IP cores + NoC = NDP
  - Scalable, portable means of building up MPSoCs



Current Collaboration

- Gilberto Contreras (Princeton PhD student) spending summer at U. Patras in Greece, working with Stefanos Kaxiras
- Main effort: Heterogeneous parallelism using NDP’s flow-oriented model
- Also: Network provisioning
- Plans: Continue telecollaboration at end of summer when Gilberto returns to Princeton

Concluding remarks

- Summary of NDP results thus far:
  - Up to 6.95X speedup for simple parallelizations on 16 cores
  - Up to 30% savings in energy-delay-product
  - 4% additional area
- Overall, Network-Driven Processing:
  - Drives application partitioning at very low hardware overheads vs. existing software approaches
  - Eases user and compiler mapping
  - Handles run-time variabilities such as power and faults
  - Ensures software portability across hardware generations
  - Ease of hardware scaling to future chip generations

An example: Quicksort mapped onto flow execution model

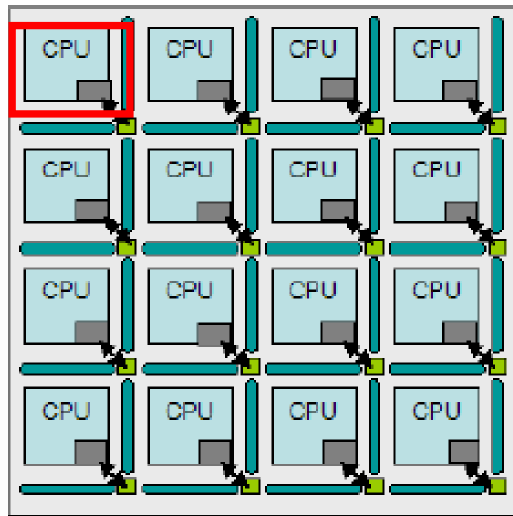
```
quicksort (int *a, int low, int high)
{
   int pivot;
   if (high>low) {
      pivot = partition(a, low,
         high);
      quicksort(a,low,pivot-1);
      quicksort(a,pivot+1,high);
   }
}

main() {
   //sort 1024 elts
   quicksort(a,0,1024)
}
```

Original

Flow Model

```
1  quicksort ()
2  {
3  int pivot;
4
5    if (high>low) {
6    {
7    _flow_consume(a);
8
9    pivot = partition(a,low,high);
10   _flow_create(quicksort);
11   _flow_produce(a,sizeof(int)*(pivot-
     low));
12
13   _flow_create(quicksort);
14   _flow_produce(a+sizeof(int)*pivot,
     sizeof(int)*(high-pivot));
15   }
16   }
17
18   main()
19   {
20   _flow_create(quicksort);
21   _flow_produce(a, sizeof(int)*1023);
22   }
```

Walkthrough

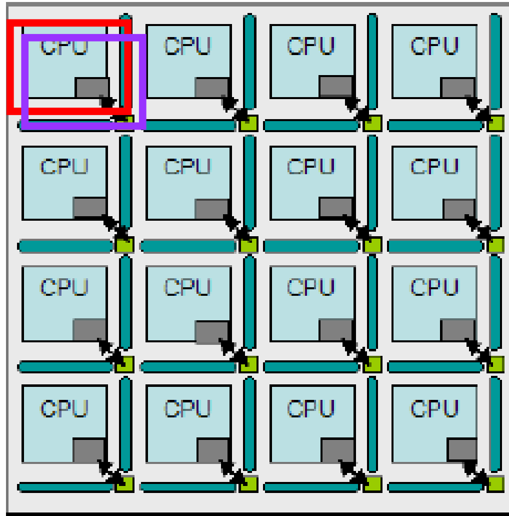



```
1   quicksort ()
2   {
3   int pivot;
5      if (high>low) {
6     {
7     _flow_consume(a);
8
9     pivot = partition(a,low,high);
10    _flow_create(quicksort);
11    _flow_produce(a,sizeof(int)*(pivot-low));
12
13    _flow_create(quicksort);
14    _flow_produce(a+sizeof(int)*pivot,
       sizeof(int)*(high-pivot));
15    }
16    }
18    main()
19    {
20    _flow_create(quicksort);
21    _flow_produce(a, sizeof(int)*1023);
22    }
```

Tile 0's Flow Table at beginning of execution:

Flow ID	PC	State	Reg Ctxt ID	Input Q Src	Output Q Dest
f0	Line 18	Ready		None	None

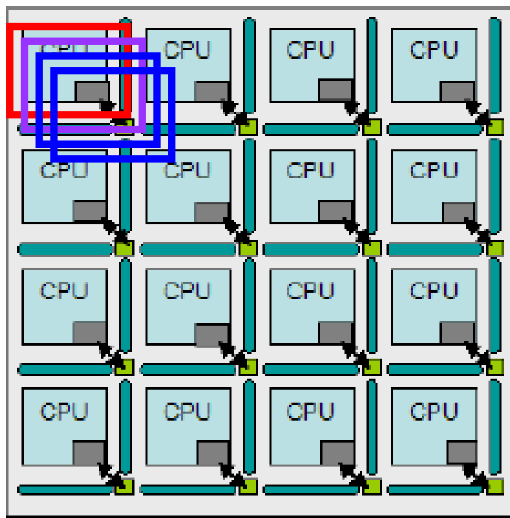
Walkthrough




```
1   quicksort ()
2   {
3   int pivot;
5      if (high>low) {
6     {
7     _flow_consume(a);
8
9     pivot = partition(a,low,high);
10    _flow_create(quicksort);
11    _flow_produce(a,sizeof(int)*(pivot-low));
12
13    _flow_create(quicksort);
14    _flow_produce(a+sizeof(int)*pivot,
       sizeof(int)*(high-pivot));
15    }
16    }
18    main()
19    {
20    _flow_create(quicksort);
21    _flow_produce(a, sizeof(int)*1023);
22    }
```

Tile 0's Flow Table after first parallel flow f1 is spawned:

Flow ID	PC	State	Reg Ctxt ID	In Q Src	Out Q Dest
f0	line18	Active	0x1	none	To f1
f1	line1	Ready		From f0	None

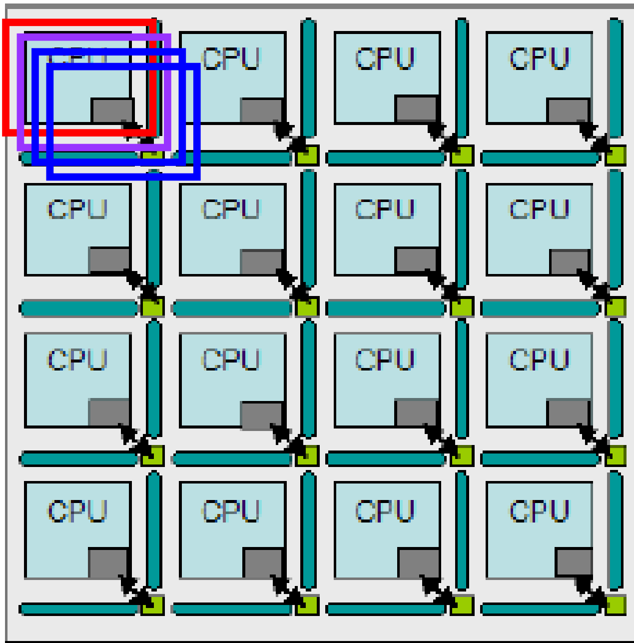


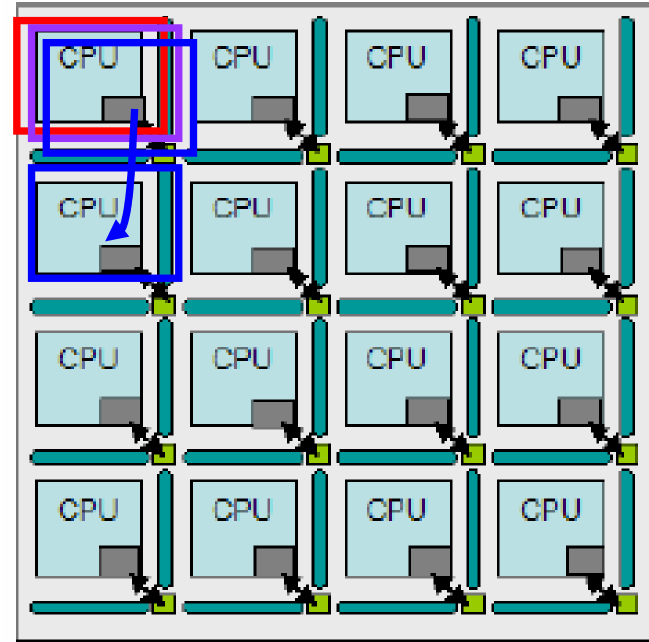

```
7     _flow_consume(a);
8
9     pivot = partition(a,low,high);
10    _flow_create(quicksort);
11    _flow_produce(a,sizeof(int)*(pivot-low));
12
13    _flow_create(quicksort);
14    _flow_produce(a+sizeof(int)*pivot,
       sizeof(int)*(high-pivot));
15    }
16    }
18    main()
19    {
20    _flow_create(quicksort);
21    _flow_produce(a, sizeof(int)*1023);
22    }
```

Tile 0's Flow Table after flows f2 and f3 are spawned:

Flow ID	PC	State	Reg Ctxt ID	In Q Src	Out Q Dest
f0	line18	Blocked	0x1	none	To f1
f1	line1	Active	0x2	From f0	To f2, f3
f2	line1	Ready		From f1	none
f3	line1	Ready		From f1	none

Tile 0: Hmm, it's getting crowded in here…





- More than just migration though…
- Establishing and tracking flow relationships lets you:
  - Reserve/optimize network bandwidth
  - Speed-balance between producer consumer
  - Optimize coherence and communication
  - Cleanly manage core failures

Walkthrough

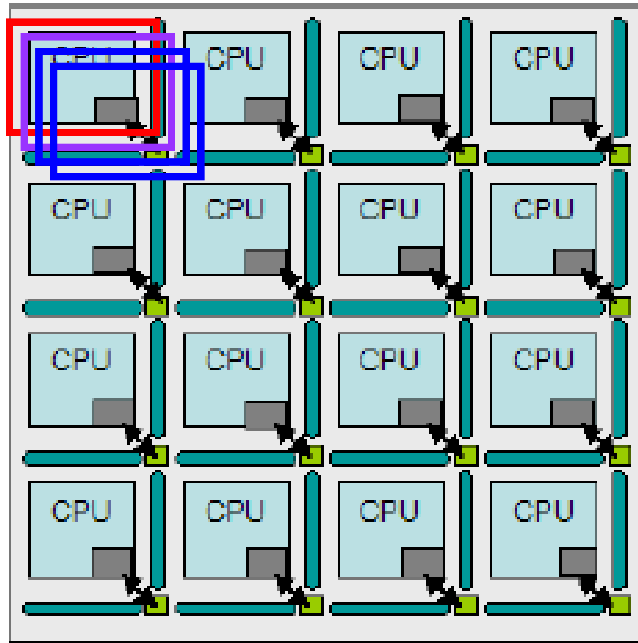
Tile 0's Flow Table after flow f3 is placed at Tile 1:

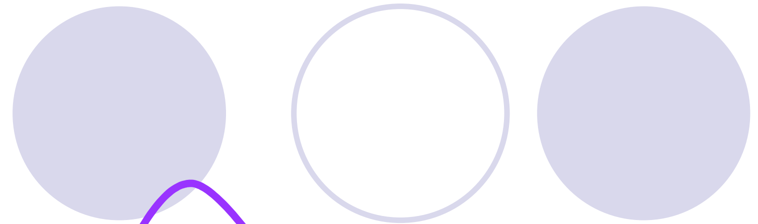
Flow ID	PC	State	Reg Ctxt ID	In Q Src	Out Q Dest
f0	line18	Blocked	0x1	none	To f1
f1	line1	Blocked	0x2	From f0	To f2: local, To f3: tile 1
f2	line1	Active	0x3	From f1	none

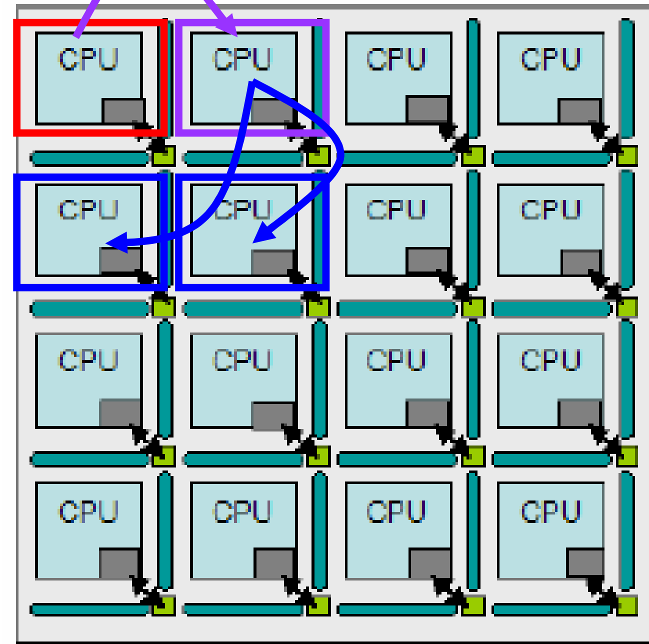
Tile 1's Flow Table after flow f3 is placed at Tile 1:

Flow ID	PC	State	Reg Ctxt ID	Input Q Src	Output Q Dest
f3	line1	Active	0x1	From f1: Tile0	None

Eventually…



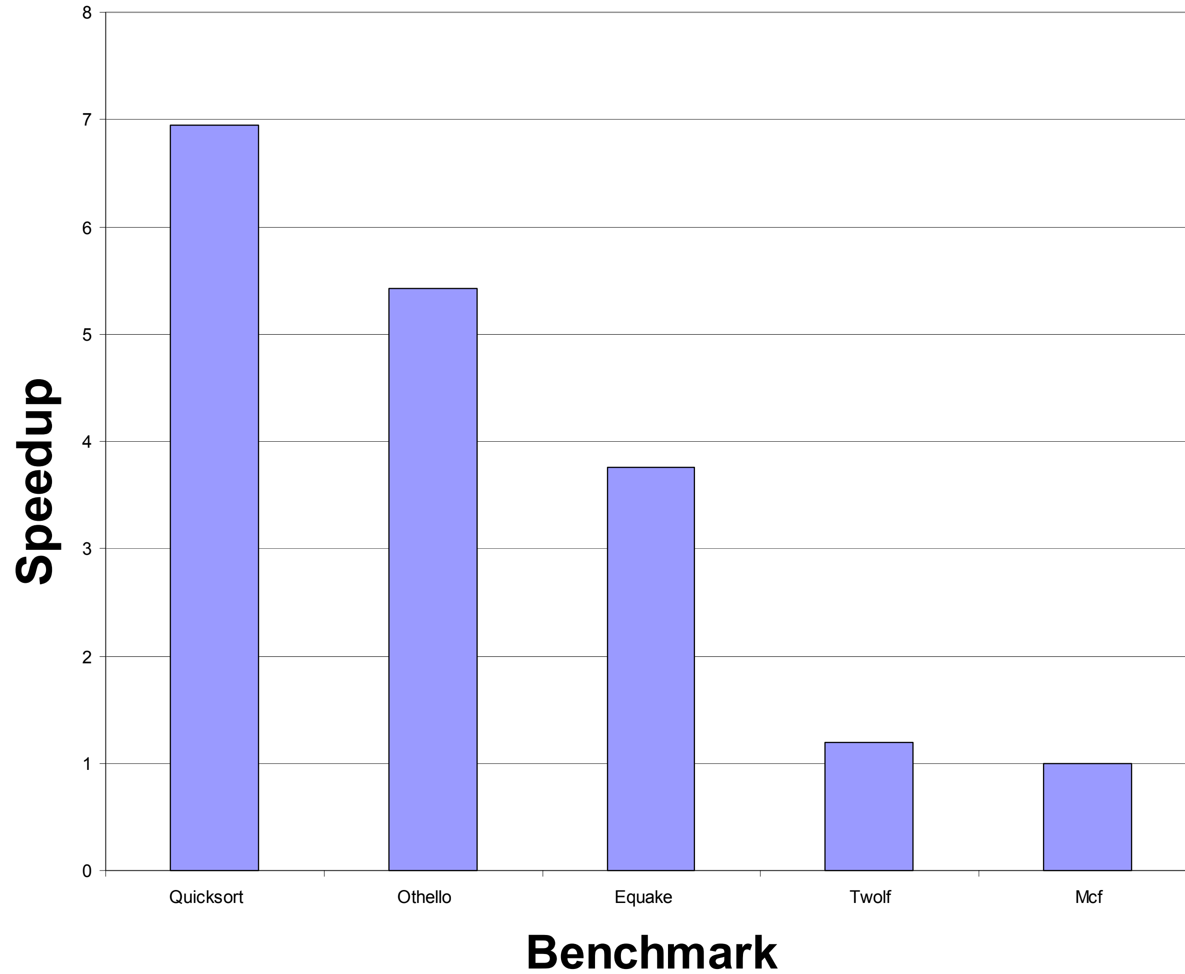




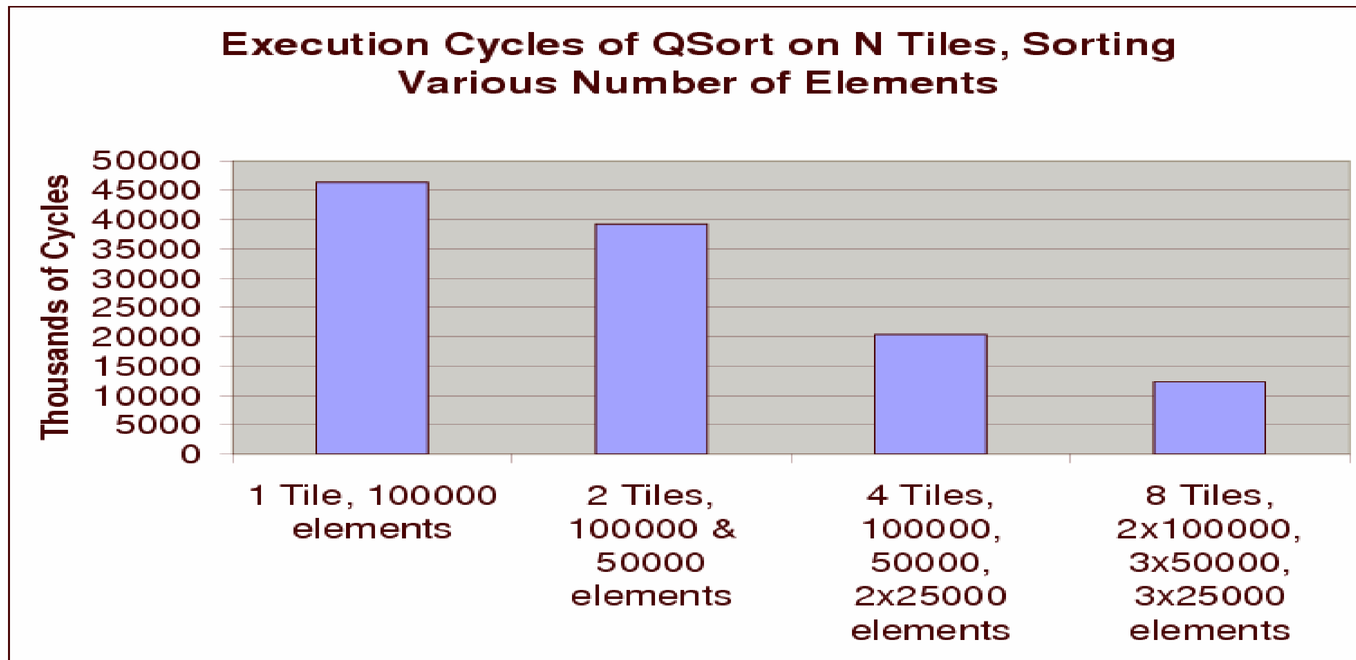
Preliminary evaluation results

- Simulator infrastructure
  - Cycle-level simulator of proposed NDP architecture
  - 16 ARM cores, 2-way superscalar
  - Additional NDP instructions (Gcc with intrinsics)
  - Simple greedy heuristic-based policy controller
  - Power models: Wattch and Orion
- Benchmarks with variable degrees of parallelism
  - Not amenable to compiler-driven application partitioning
  - quicksort, othello, equake, twolf, mcf

Speedup vs. single core

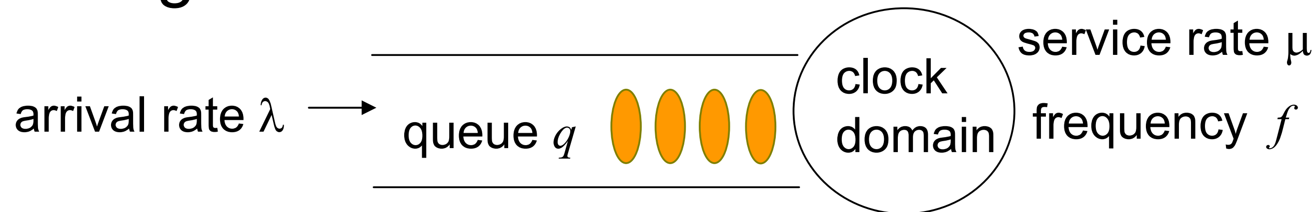


Load-balancing multiprogrammed workloads



Challenge #2 – Towards formal, stable, distributed scheduling of threads

- Basis: Formal, stable, distributed power management of threads



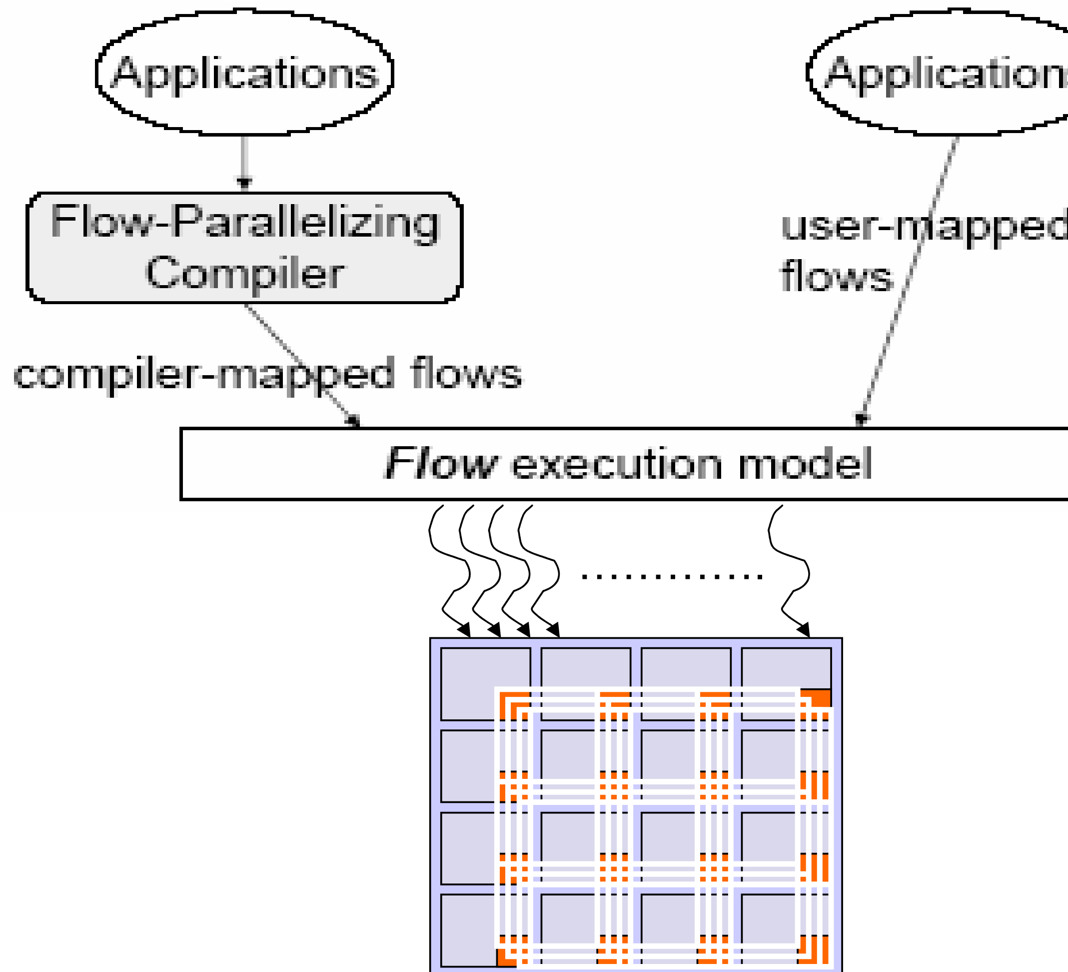
average queue occupancy for (k-1)th control interval

queue occupancy at the beginning of (k-1)th control interval

$$\overline{q'}_{k} = \overline{q}_{k-1} + \frac{T}{2}\left(\overline{\lambda}_{k-1} - \frac{1}{t_{1} + \frac{C_{2}}{f_{k-1}}}\right)$$

average queue changes due to different demand and service rates

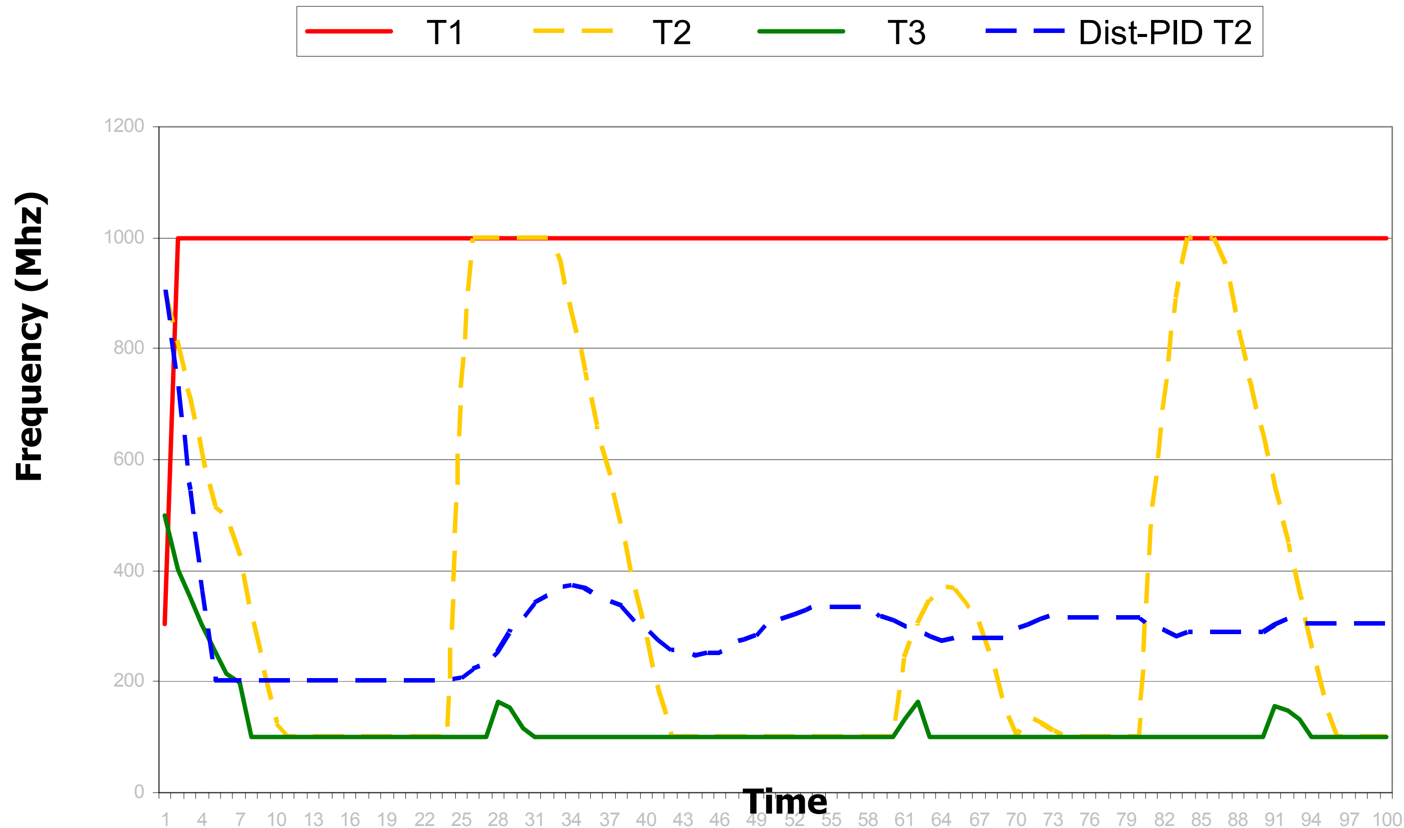
Proposed: Hardware-assisted partitioning



Map ***potential*** parallelism aggressively onto many fine-grained threads (flows), without regards for load balancing

Hardware determines ***actual*** parallelism by placing flows on cores at run-time

Dist-PID manages oscillation/bursts better than Local approaches in a CMP



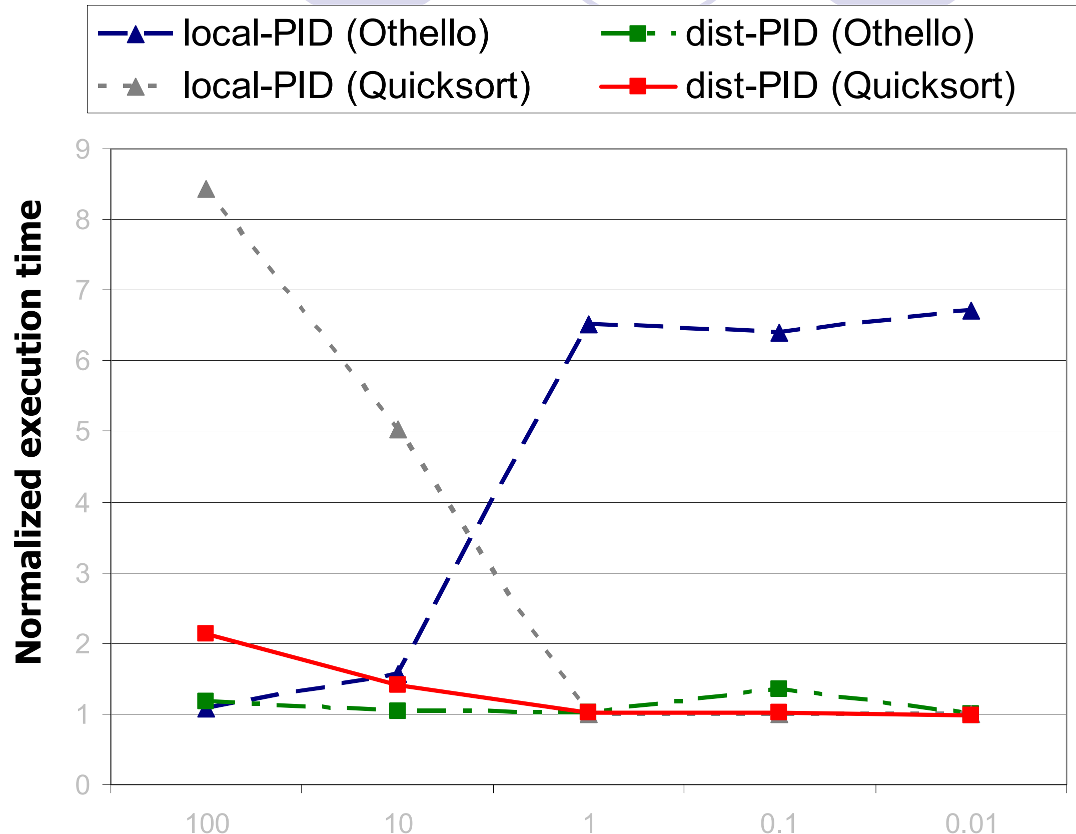
- Because of the communication, Dist-PID knows what speed to target
- Formal approach causes controller to gently zero in on optimal speed

Downsides of compiler-driven partitioning

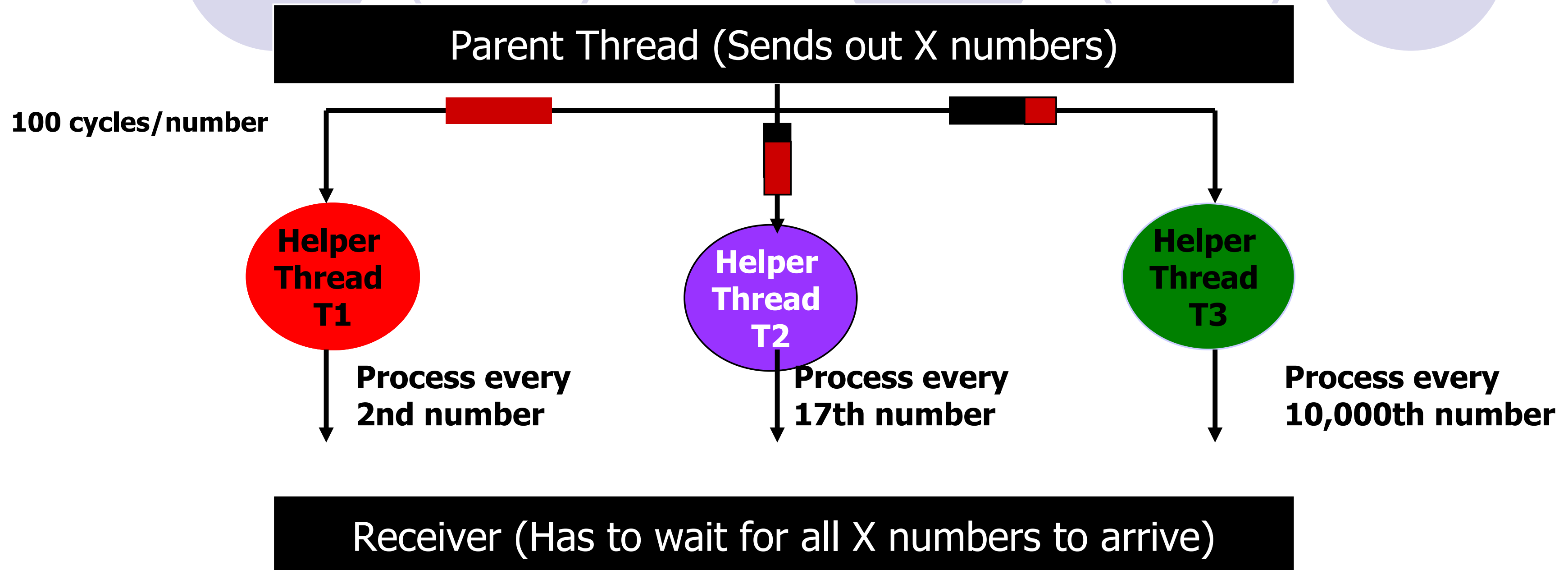
- Hard enough to extract parallelism
- Have to load balance as well
- Cannot adapt to run-time variability
  - Unpredictable data sets
  - Unpredictable memory behavior
  - Multiprogramming
  - Power/thermal hotspots
  - Faults

Dist-PID resiliency: Demonstrating stability

- Dist-PID: More resilient than local approaches to error in processor load predictions
  - Othello, quicksort



Parallel Code and DVFS : An Example



- When one input buffer fills, Parent thread stalls
- Observation 1: Thread T1 has most work to do
  - Threads T2 and T3 can run more slowly
- Observation 2: All threads (especially T2 and T3) have bursty work requirements)
  - Must avoid oscillations

Options for CMP DVFS Policies

- Static DVFS settings for whole application:
  - Based on profiling or application knowledge
  - Pro: simple, no overshoot or oscillation
  - Con: hard to gather application knowledge, especially for dynamically-varying parallel applications.
- Locally-controlled, uncoordinated V/f settings per core
  - Pro: simple, fast, easy to scale
  - Con: doesn’t account for inter-thread relationships
- Coordinated cross-chip control of DVFS settings
  - Pro: more realistic, more flexible
  - Con: Slower, possibly harder to scale
    - Which info to transfer and how fast?

Introducing Dist-PID for power management

1) Determine critical path using equation :

$$q_{target} = (K_p(q_k - q_{k-1}) + K_i q_k - \mu_k + \mu_{k-1})/K_i$$

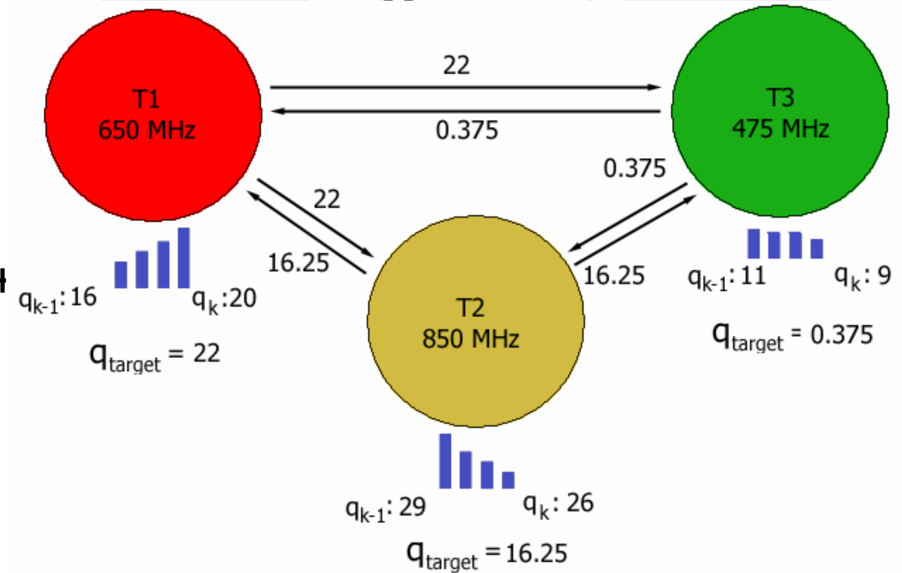
2) Distribute to all processors

Exchange q_{target} between processors

Choose highest q_{target} seen: this is critical path

3) Use highest q_{target} as new q_{ref} and solve equation

$$\mu_k = \mu_{k-1} + K_i(q_k - q_{ref}) + K_p(q_k - q_{k-1})$$



- Intuitively:
  - Who is the critical path?
  - To preserve performance, run that processor at maximum speed
  - To save energy, run everyone else slower