Partial Paging for Real-Time NoC Systems

Adrian McMenamin, Professor Neil Audsley
Real Time Systems Group,
Department of Computer Science, University of York

Virtual memory: why bother?

Familiar, powerful, paradigm:

“The value of a computer system to its users is greatly enhanced if a user can, in a simple and general way, build his work upon procedures developed by others. The attainment of this essential generality requires that a computer system possess the features of equipment-independent addressing, an effectively infinite virtual memory, and provision for the dynamic linking of shared procedure and data objects.”

Virtual Memory, Processes and Sharing in Multics (Daley, Dennis, MIT, 1968)

In real time and NoCs:

- Dynamic paging not suitable for all use cases
- But should not have to run a full OS on every node – should be able to share

The “Many-Core Age”

Shift to “many core”:

Single fast chip designs no longer feasible, bus based designs limited, so move is towards “many core” systems – Intel demonstrated 256 core NoC in 2014, Tilera a 100 core NoC this year

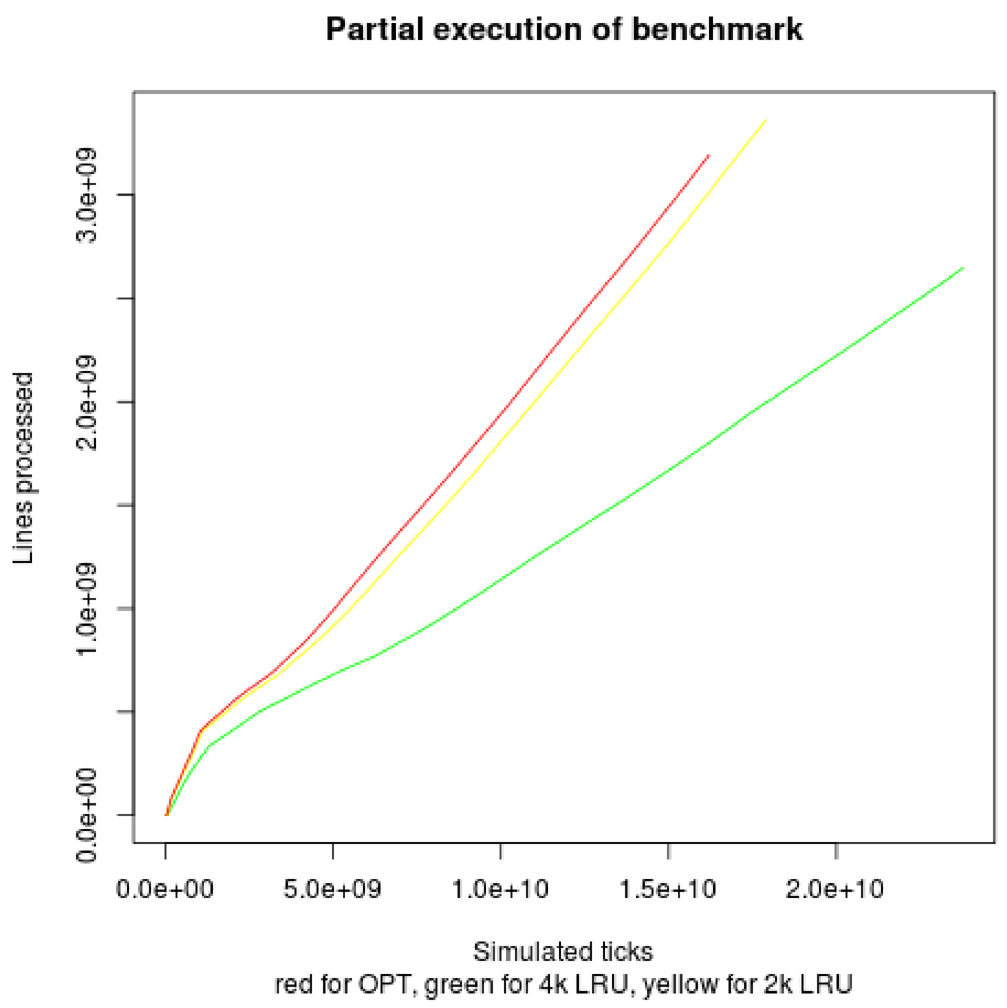
Problems:

- Familiar issue of “Amdahl's Law”
- Parallel Programming is Hard
- Imbalance between small amounts of fast local memory and large amounts of slower global memory

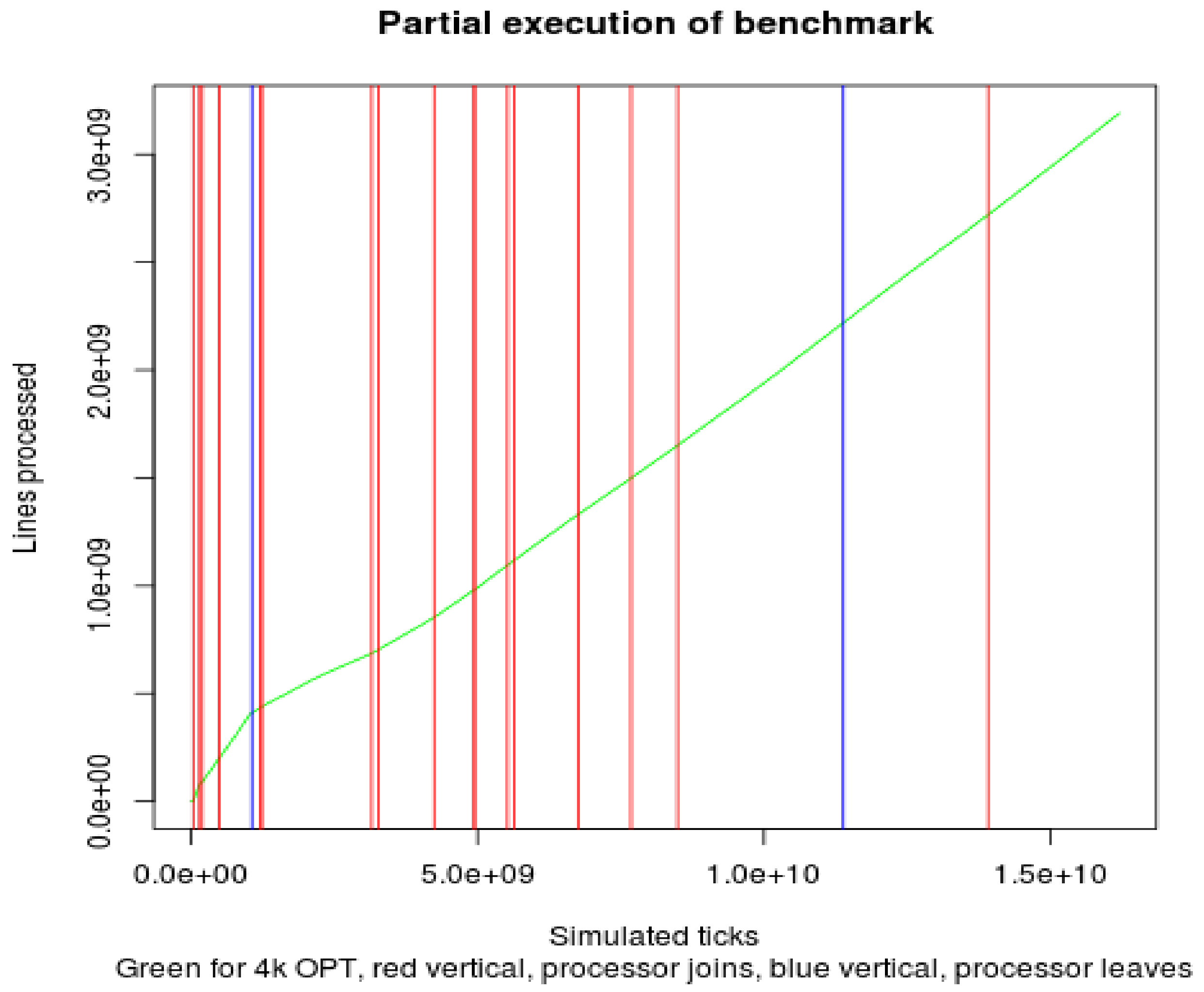
Many core + VM = thrashing

Use x264 benchmark from PARSEC to get full memory trace across multiple threads – then use this to model the behaviour of a “NoC”: assuming local memory is 1 cycle away, and global memory is 100 cycles per 16 byte cache line away.

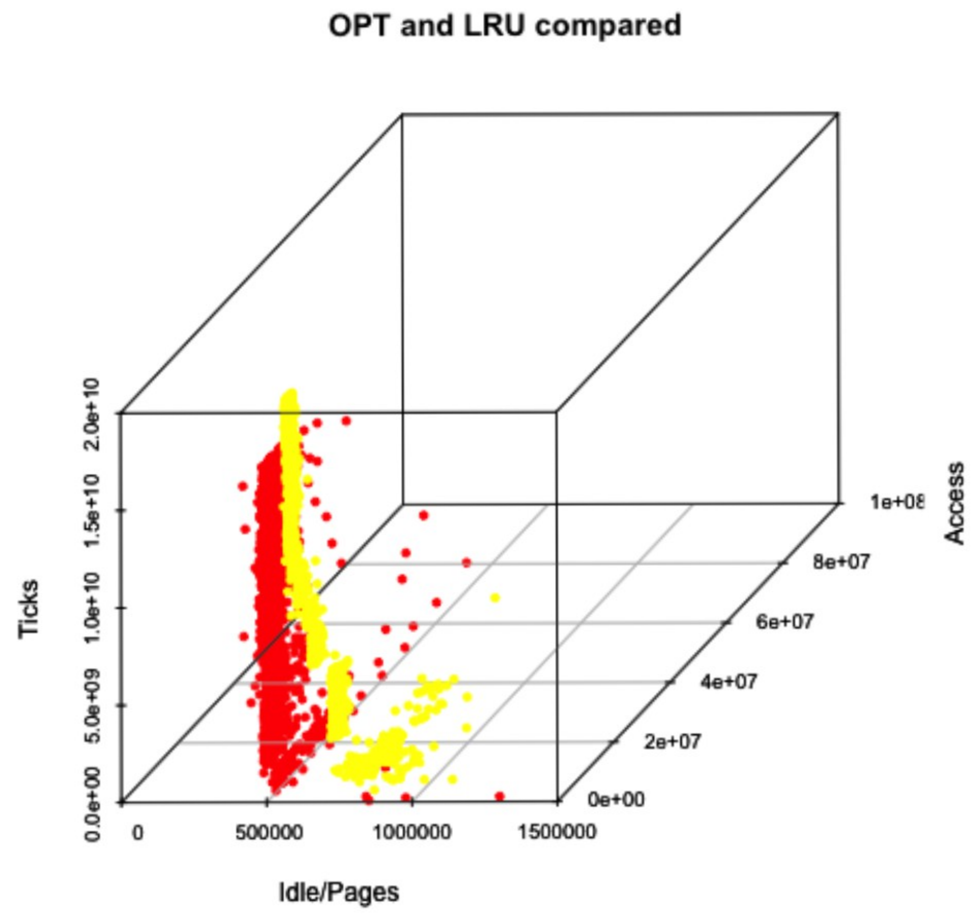
Even for OPT thrashing characteristic seen.



Closer look at thrashing

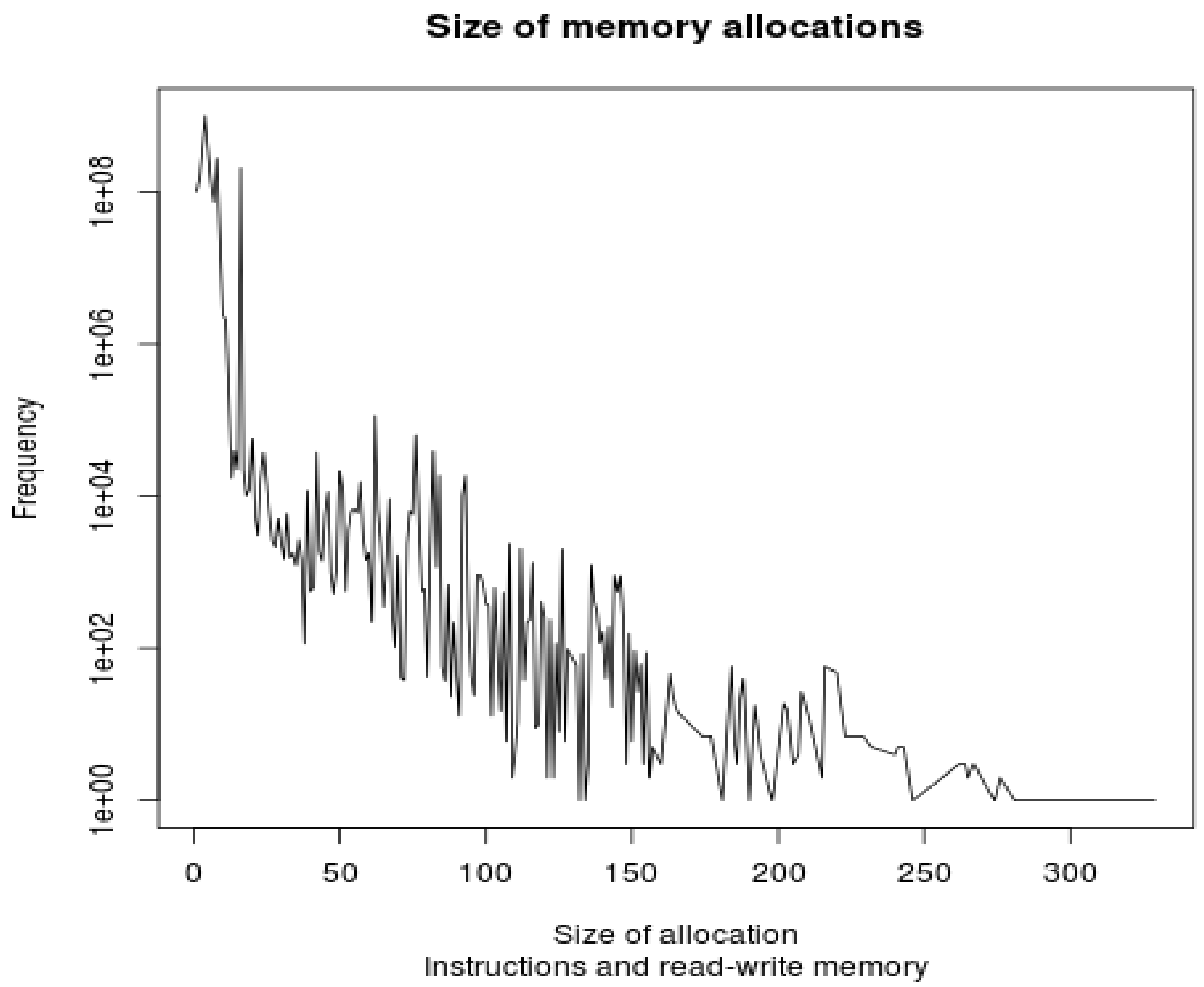


But it's not like "last time": issue is not page replacement



OPT in red, LRU in yellow

Under the hood: allocation sizes



Lessons learned?

- If we want efficient VM in multicore/NoCs we cannot use traditional paging algorithms
- But smaller pages are more efficient for memory starved systems
- Most allocations are small – smaller even than the smallest page size

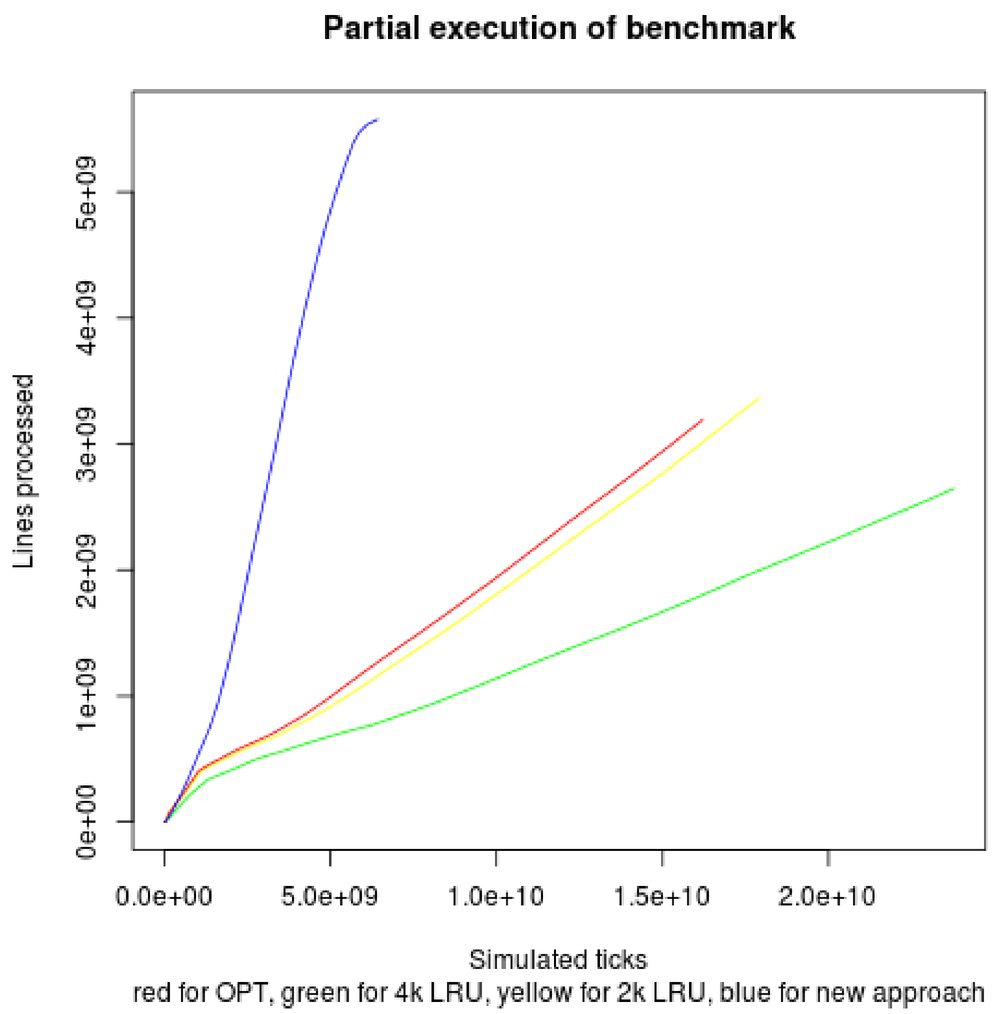
Trying a partial page

- Preserve the advantage of VM – shared address space – but do not wait around loading memory we never use.

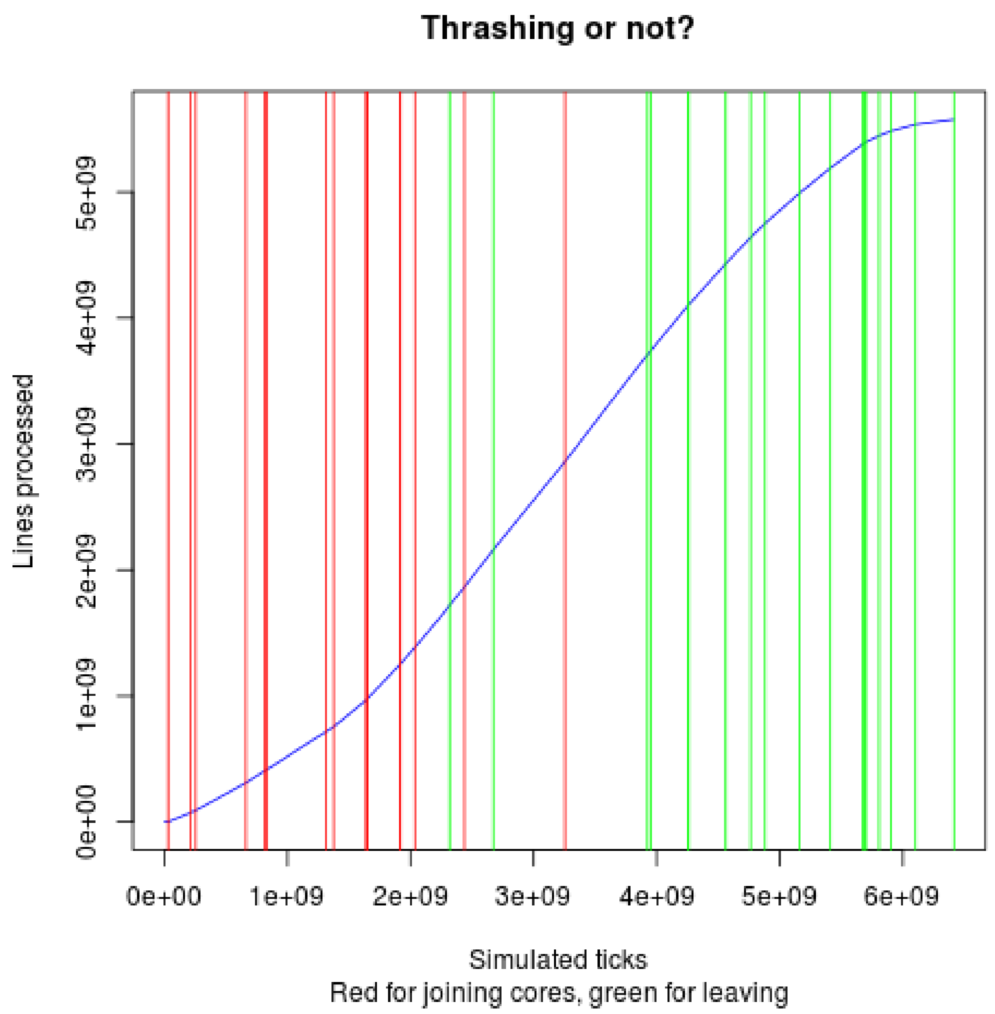
- Pragmatic balance between smaller allocations and existing technologies and well-understood techniques.

- Test allowing for some increased processing time

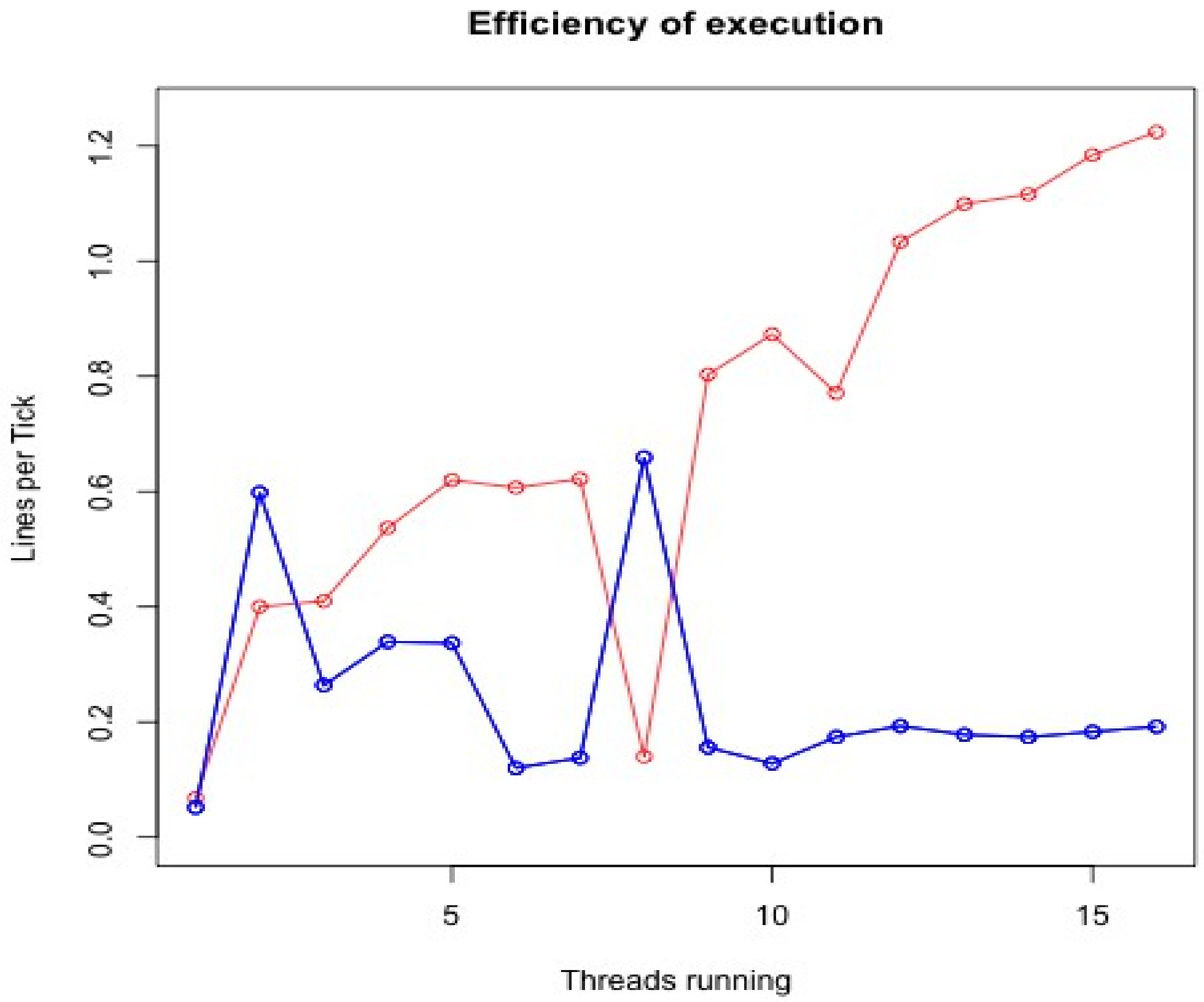
Results seem promising



Thrashing mitigated



Efficiency compared

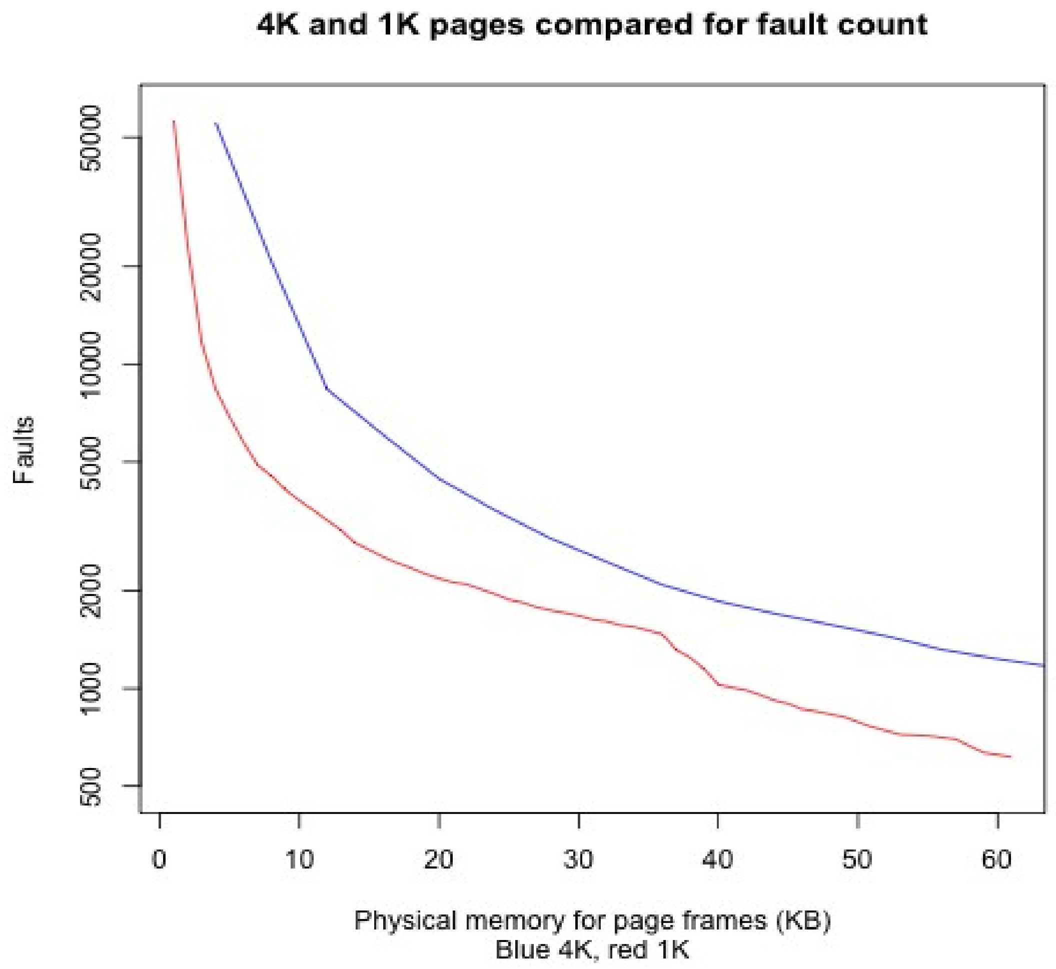


A more thorough test

- A number of assumptions in the model: about time taken to process “partial” paging and memory availability in particular
- Need to test the validity of these on a system closer to the real world – used OVPSim Microblaze: instruction accurate simulator of a software core that can execute one instruction per cycle
- By changing the model we can simulate hardware innovations (such as MMU supporting partial paging)
- But can only test on a single core – OVPSim cannot model multiple asynchronous cores

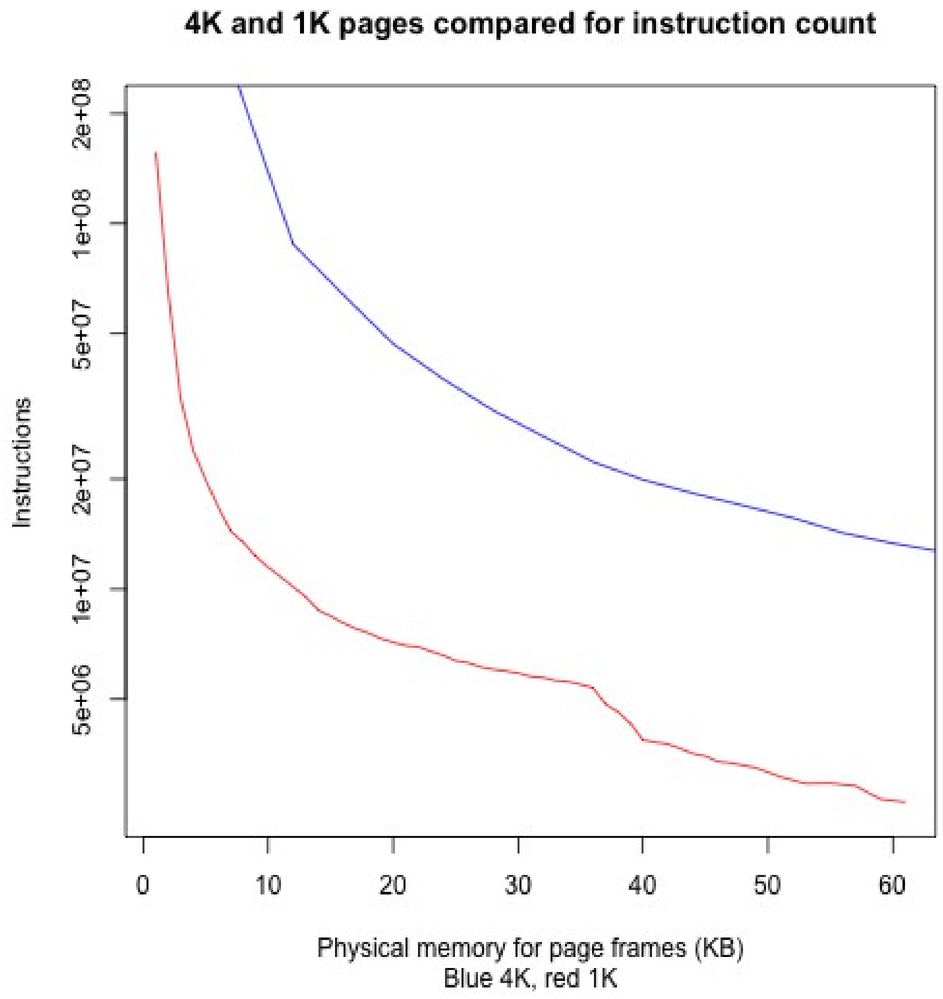
Establishing the baseline

Using traditional paging, can see smaller (1k versus 4k) pages perform better on our simple test load.



Factoring in instruction counts

There is no immediate DMA support available, so pages have to be loaded "by hand": though this is more deterministic it adds to the cost of 4k pages.



How does partial paging do?

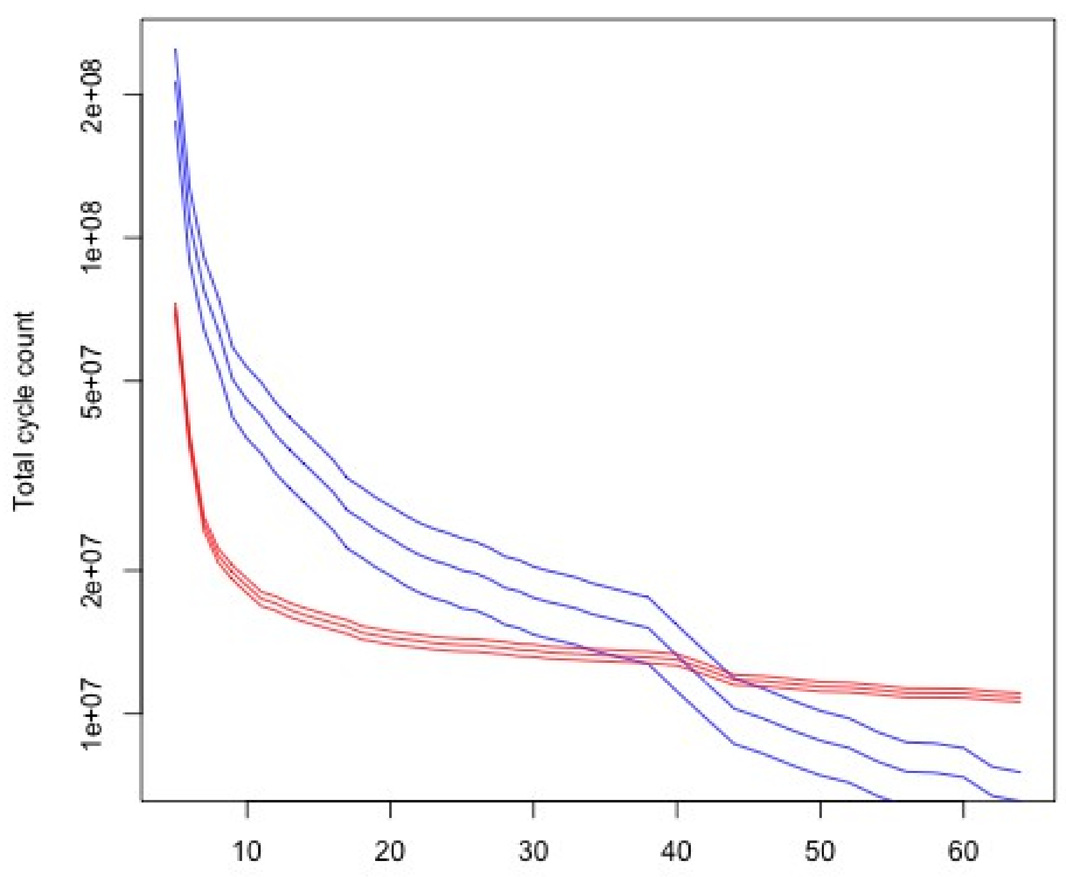
Lower instruction count at only very small memory numbers…

TLBs	Instructions: traditional paging	Instructions: partial paging
8	20.2 million	18.5 million
16	9.9 million	13.5 million
24	7.5 million	12.3 million
32	6.5 million	11.8 million

But when normalized for time

Reading and testing bitmaps is expensive, but if we normalize for time taken to load pages, partial paging performs better.

Estimated cycle counts



TLBs in use
Blue for traditional, red for alternative

Some further tests

- Is 16 byte size too small? (Hard fault numbers constant, but “small” faults decrease for larger sizes, but by less than additional instruction cost)
- Using FIFO (no time source) – could LRU do better? (Not on simple system)
- What about other loads? (Evidence suggests locality a key factor in performance)

Further research

- Will it work for 256, 512 and 1024 cores?

- No consideration of problems of consistency and coherence in multicore – an essential next step.

- Is sub-cycle hardware for this task feasible? Need to validate the approach.