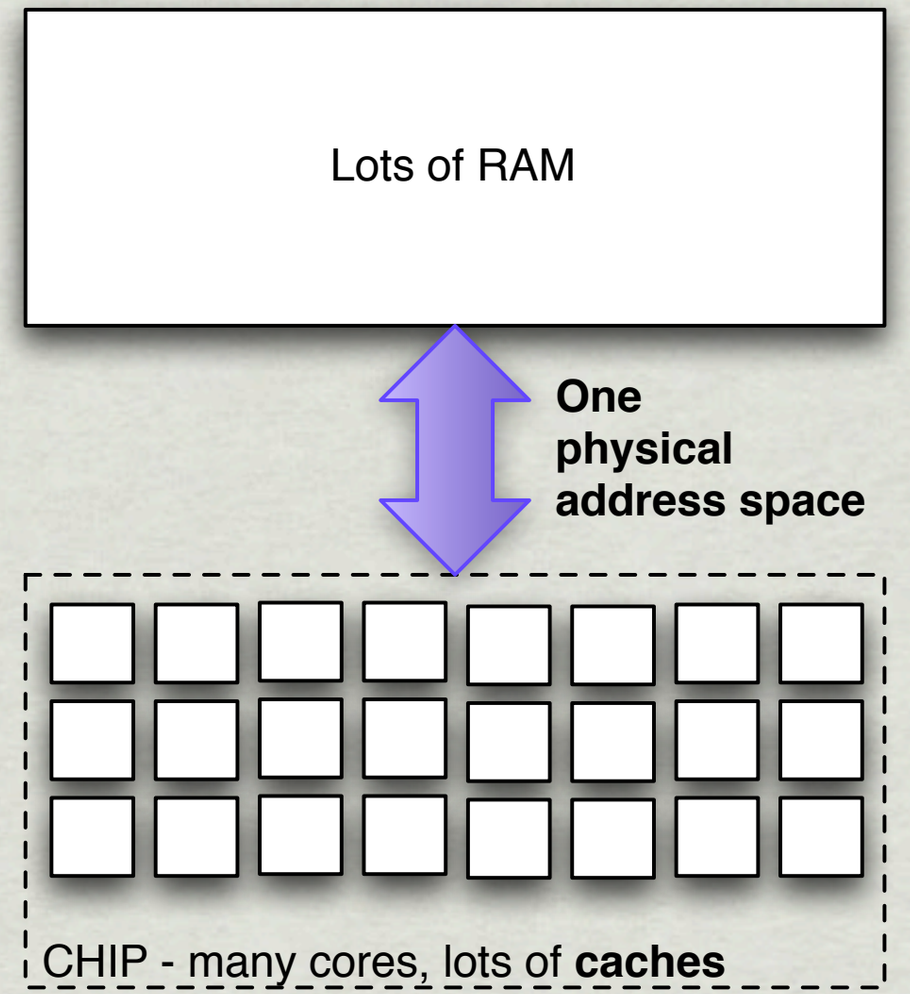
# Thread-local storage: The bane of shared memory many-core architectures

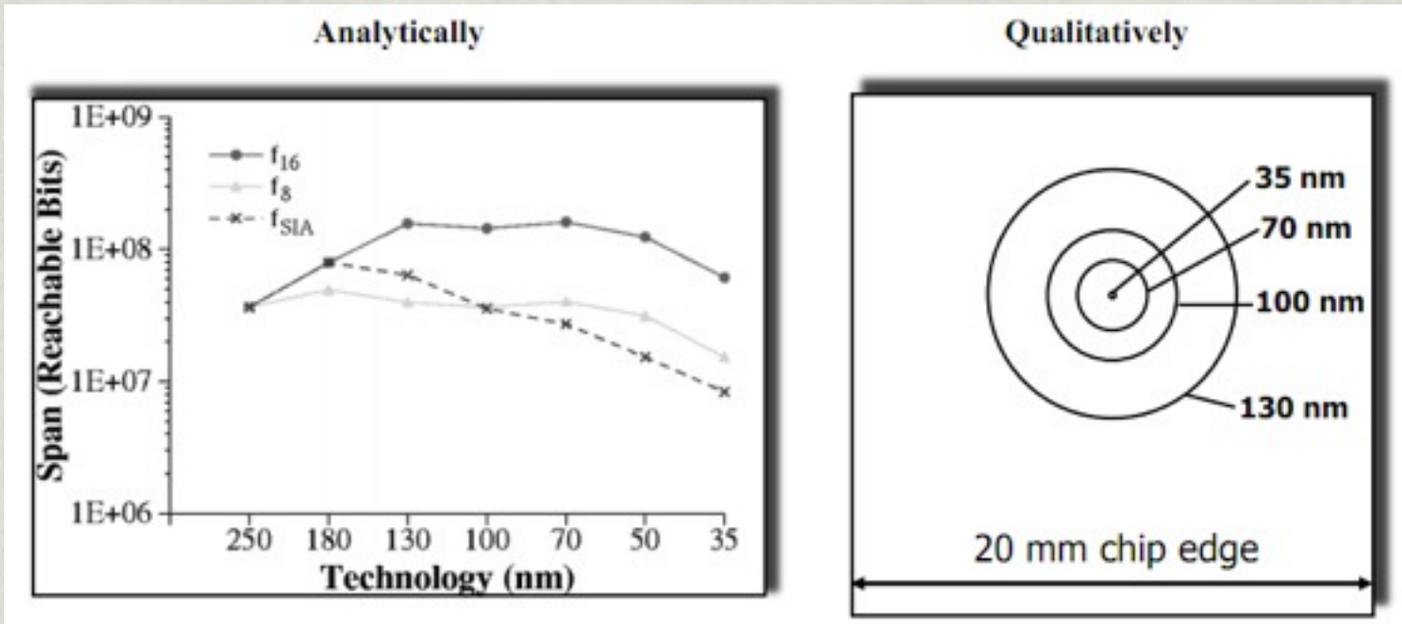
kena - 20110928

# What is a shared memory many-core?



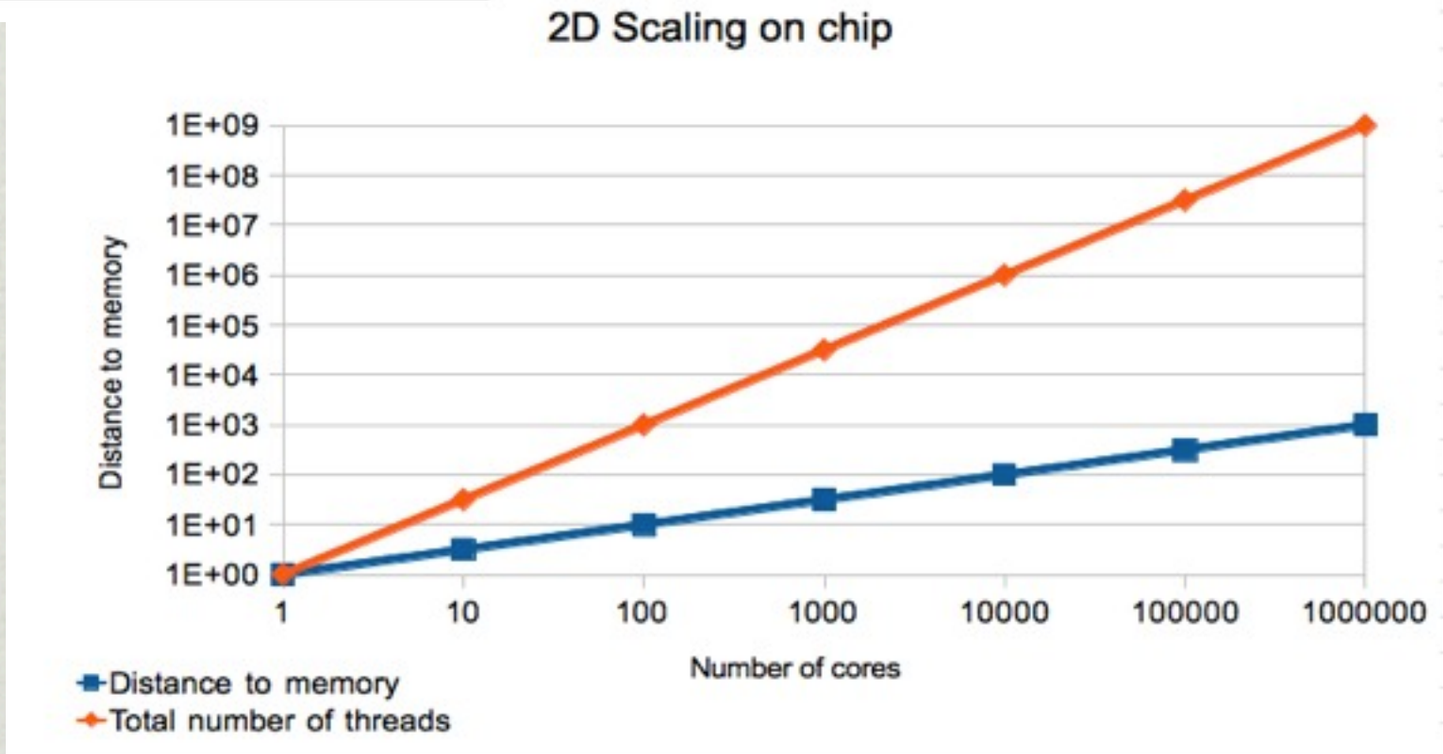

- "Many" **processing units** (cores, pipelines, FUs)
- All processing units share **a single memory system**
- **Memory** = **storage** with **load/store interface + addresses**
- Memory system = network of caches with a **single physical backing store**

# Latency issues




**DENSER CHIP**
**= MORE CLOCK CYCLES**
**TO REACH DATA**
**(AT CONSTANT FREQUENCY)**

**MORE CLOCK CYCLES**
**TO REACH DATA**
**= MORE THREADS PER**
**CORE NECESSARY TO**
**TOLERATE THE LATENCY**




**DATA: ARGAWAL&AL 2000, JESSHOPE&AL 2010**

# Threads vs tasks vs workers vs processes

- “Thread” applies to:
  - “**hardware threads**” when each hw thread has a sw scheduler to time-share processes
  - “**worker threads**” when each worker runs computational tasks one after each other without interleaving
  - “**microthreads**” when each hw thread runs exactly one task (and can be recycled)
  - NOT the time slice in time-sharing, NOT the task in worker-based scheduling

# Threads vs tasks vs workers vs processes

- What kind of “thread” are we talking about?
  - processing unit that **computes** one or more **functions** sequentially
    - initial state: which **function**(s), which **input**
    - private state: one or more **activation records**
- Threads are just **sets of activation records** defined **simultaneously** - TLS ≈ activation records

# What is TLS?

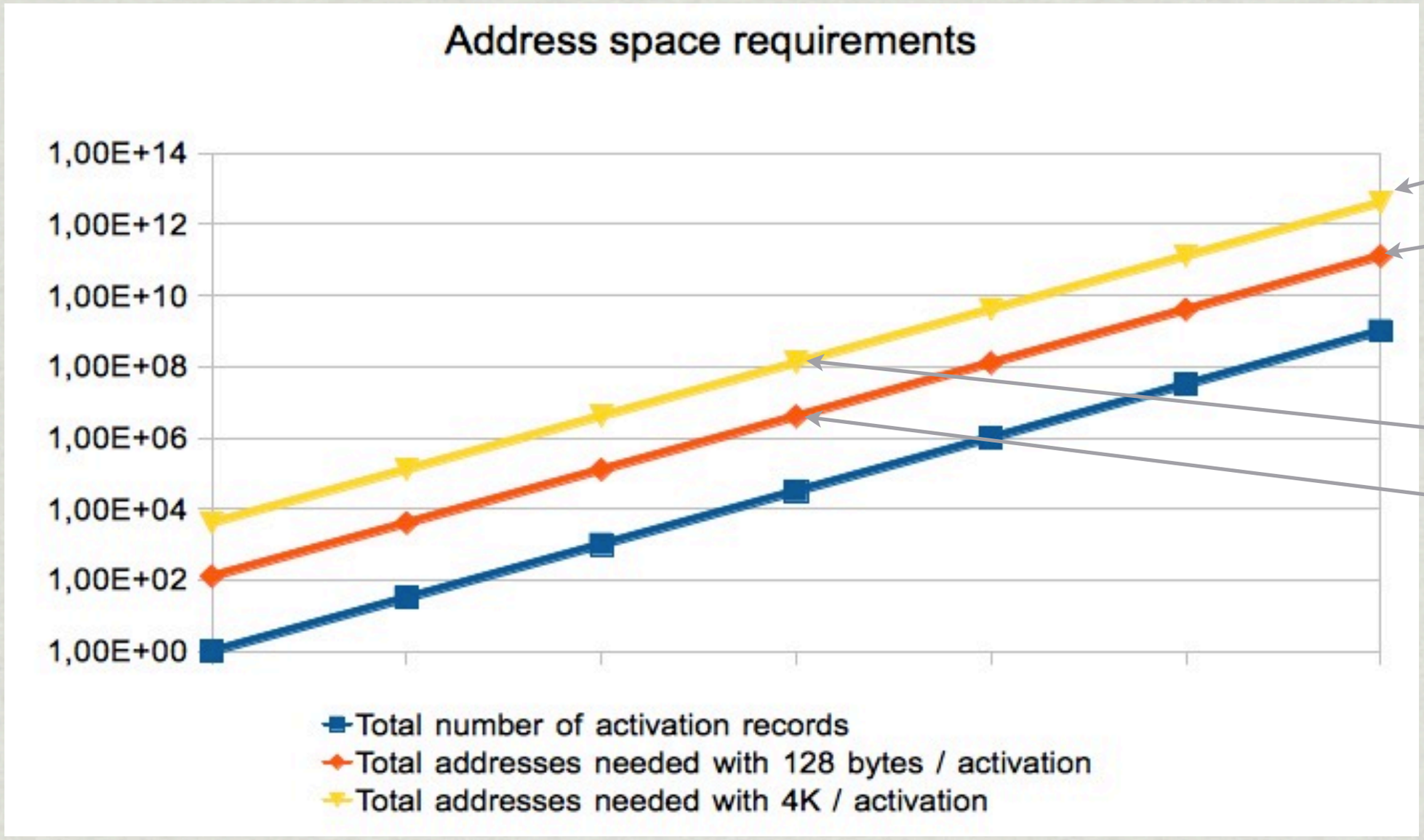
- **WHAT**: “private to each thread”
  = *a priori* **knowledge that the locations in memory are not shared**
- **WHY**: necessary for Turing-completeness
  - **activation records** = return addresses + local variables
- The more TLS there is, the more Turing-complete the threads are
  - Current GPU “threads” do not have a large TLS and thus are mostly limited to primitive-recursive functions, *ie* not much
- Applied halting problem: if the computation contains data-dependent recursions (all non-primitive-recursive functions do), **it is not possible to determine before a thread start how much TLS it will need** to complete

# TLS with shared memory: address spaces

- Only two situations really:
  - **Single address space** (no translation)
    ⇒ partition the address space
  - Multiple **virtual address spaces** with translation
    ⇒ partition the physical address space

    to construct the translation entries
- **Dynamic address space partitioning is the issue**
- Using “malloc” to allocate ≈ also dynamic partitioning

# Why global static partitioning won't work




4,3 PB

140 GB

130 MB

4,1 MB

THE CELLS MUST BE ON CHIP TO ENSURE LOCALITY

# Diversity of sizes

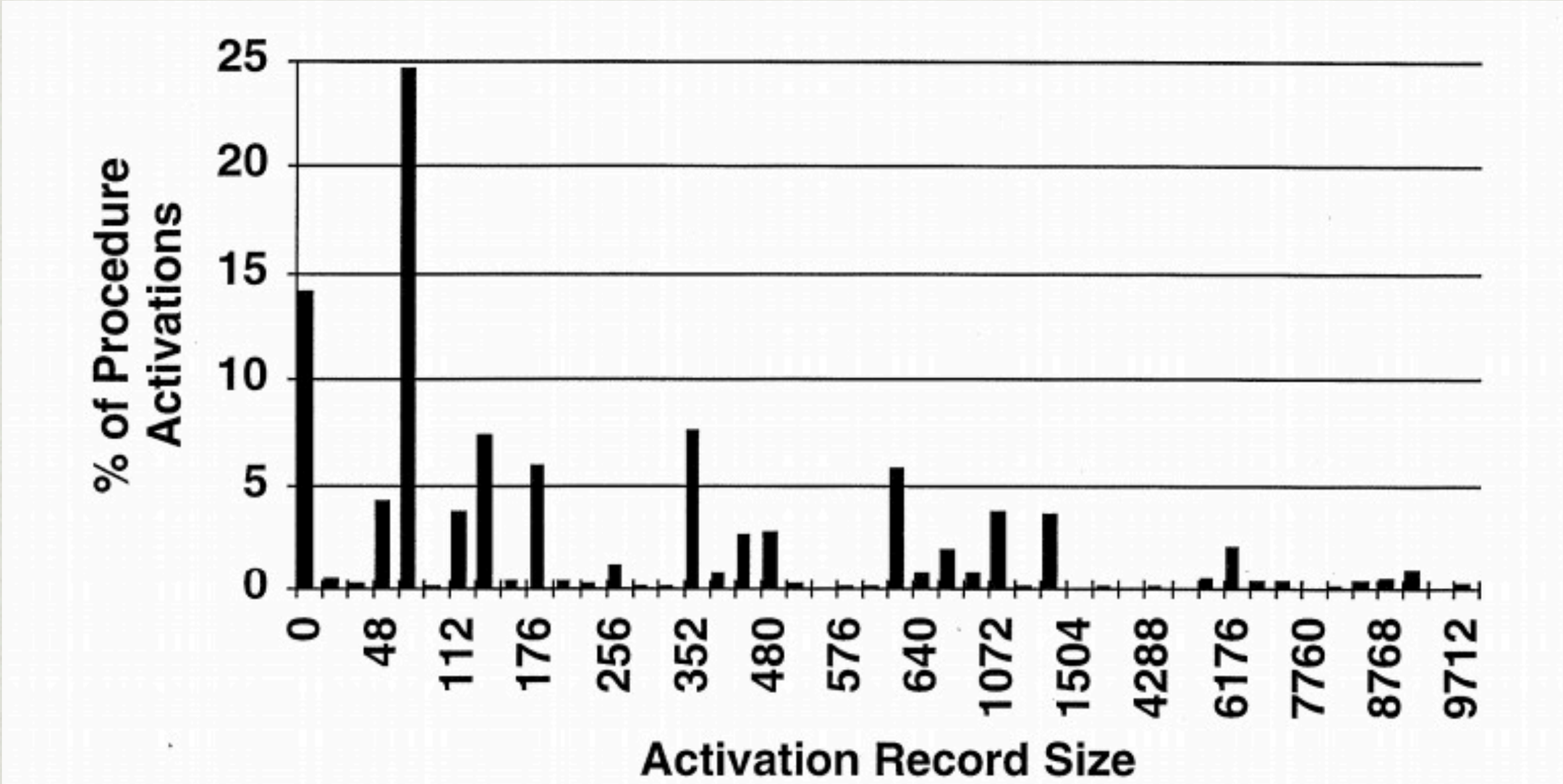
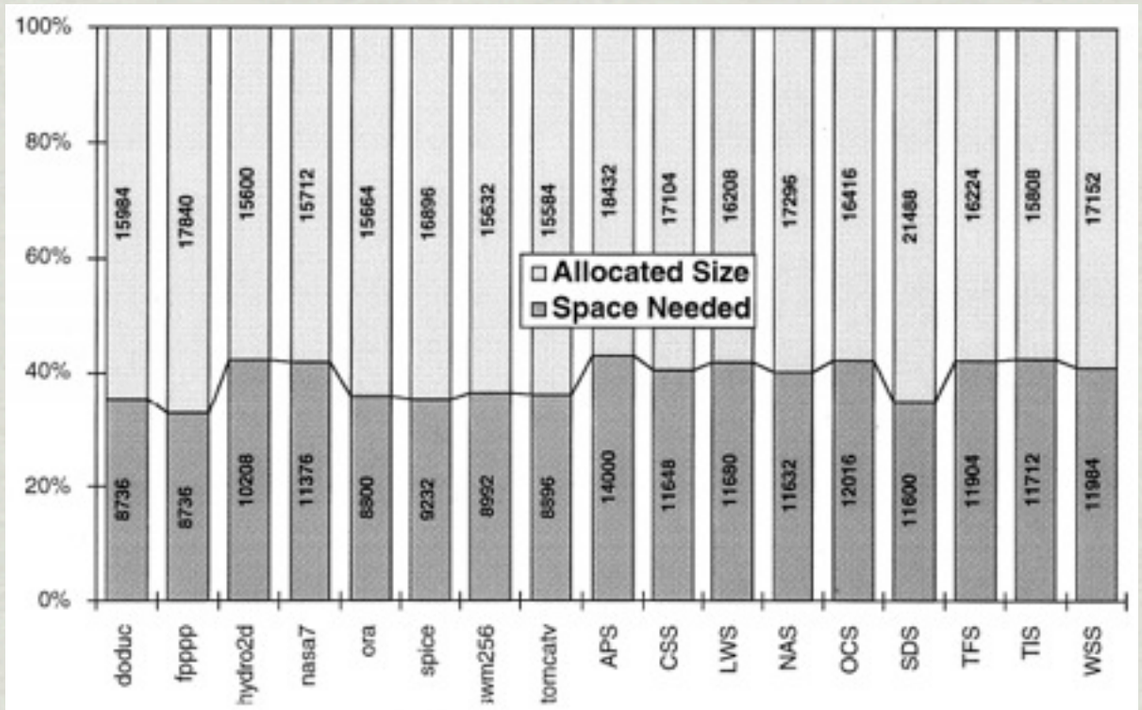
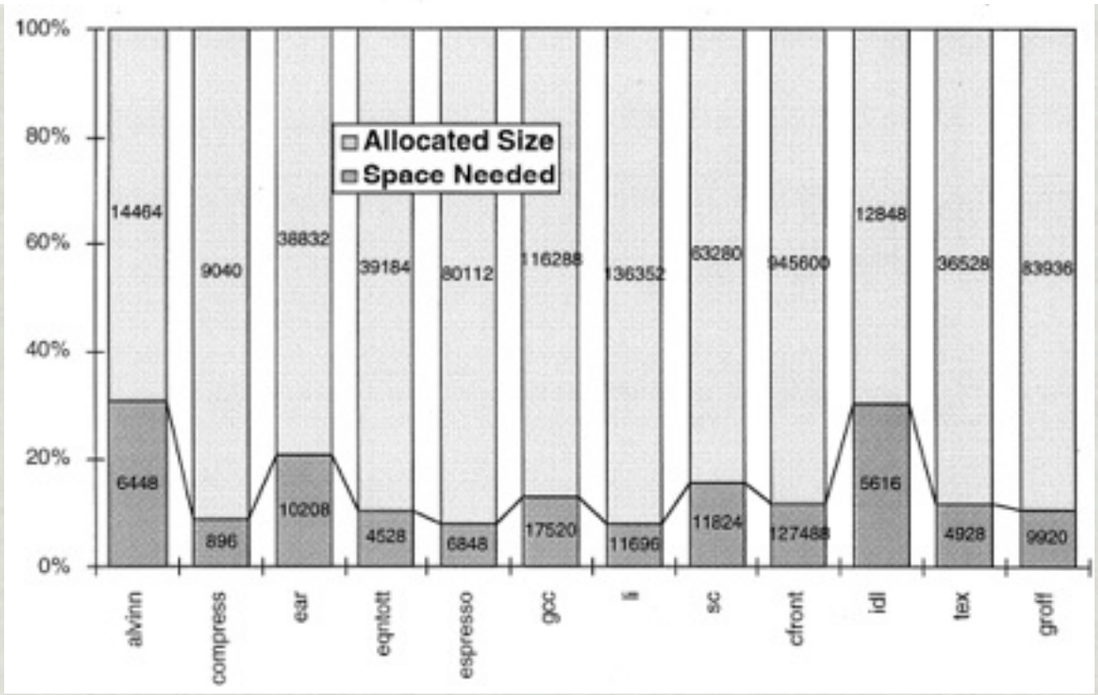



**Figure 2**: Fraction of activation record sizes for single run of Gnu C compiler. Activation records sizes larger than 10,000 Bytes have been eliminated, as have sizes contributing less than 0.01% of the total activation record.

DATA: GRUNWALD 1994

# Limits of pre-allocation




- Pre-allocation sometimes possible, using compiler-known frame sizes and profile data
- Actual required size usually much lower than conservative static estimate
- ⇒ dynamic re-sizing

  required **during execution**

**DATA: GRUNWALD 1994**

# Shared memory management

- Wilson 1995: *Memory management is where the rubber meets the road--**if we do the wrong thing at any level, the results will not be good**. And if we don't make the levels work well together, we are in serious trouble. In many areas of computer science, problems can be decomposed into levels of abstraction, and different problems addressed at each level, in nearly complete isolation. Memory management requires this kind of thinking, but that is not enough--**it also requires the ability to reason about phenomena that span multiple levels.***

  ***(P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In Memory Management, LNCS volume 986, pages 1–116.)***

# Scalable memory management

- Berger & al, 2000: Hoard “state of the art scaling”
  - Uses one local heap per per sequential unit of execution (processor, worker, etc. *ie* our “threads”)
    - **A(t) = O(U(t) + P)**

(A = TOTAL FOOTPRINT, U = ACTUALLY USED, T = TIME)

- ⇒ overall address space requirement super-linear in P
- Anything less incurs contention and kills latency tolerance
- Same for ptmalloc, mtmalloc, dlmalloc and others

# Reductio ad absurdum

- Locality and caches only reduce latency of access, not latency of setup - the real problem is **efficient dynamic partitioning** of a single physical storage into TLS area
- Any solution with a **single physical address space** requires partitioning of large address spaces chunks to clusters and segregate TLS management, ie **hierarchical TLS management**
- **BUT:**
  - The higher in the hierarchy, the more difficult it becomes to grow or shrink the TLS address space reserved at that level
  - At the highest levels, **TLS addresses are mostly statically defined** as a function of the average activation record size and the number of records
  - **We know that does not scale as the number of cores increases**

# Summary / conclusions

- Many cores
  ⇒ many threads required to tolerate latencies

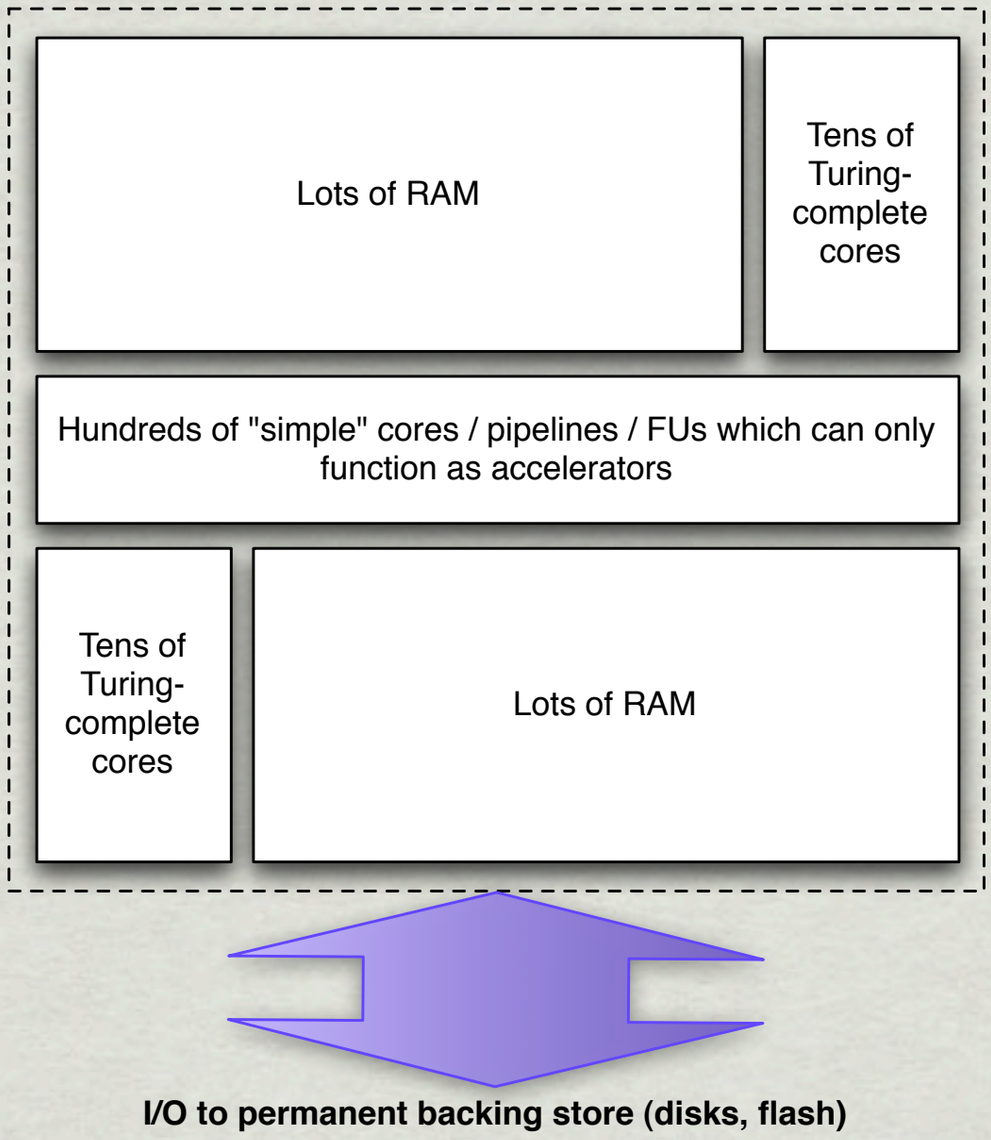
- 1 physical address space + many threads
  ⇒ hierarchical TLS management
  ⇒ mostly static partitioning at the highest levels

- static partitioning + many threads
  ⇒ impossible to fit on chip

- ***(I so hope I got this wrong... Please scrutinize!)***

- **Alternative**: registers/scratchpads = separate, local address spaces

# Possible chip structure




- Use increasing transistor counts for main RAM, not cores
- Keep cores close to RAM - on the same chip
- Different physical ASs for different core clusters (“lots ‘o scratchpad”)
- Small number of Turing-complete cores compared to accelerator cores at a single point in time
- Open questions:
  - **ratio RAM / cores?**
  - **ratio simple / complete cores?**