Adaptive proxies: handling widely-shared data in shared-memory multiprocessors

Sarah A M Talbot

Oceanography Lab, T H Huxley School of the Environment

Paul H J Kelly

Software Performance Optimisation Group, Dept of Computing

Imperial College, London

Large-scale shared-memory

- Future large-scale parallel computers *must* support shared memory
- Processors rely on cache for performance, so *scalable* cache coherency protocol needed - CC-NUMA
- Existing implementations have been plagued with performance anomalies

CC-NUMA performance anomalies

- This talk is about a simple scheme which
  - fixes various performance anomalies in CC-NUMA machines
  - without compromising peak performance
- What performance anomalies?
  - Home placement: in which CPU's main memory should each object be allocated?
  - Contention for widely-shared data: what happens when every CPU accesses the same object at the same time?

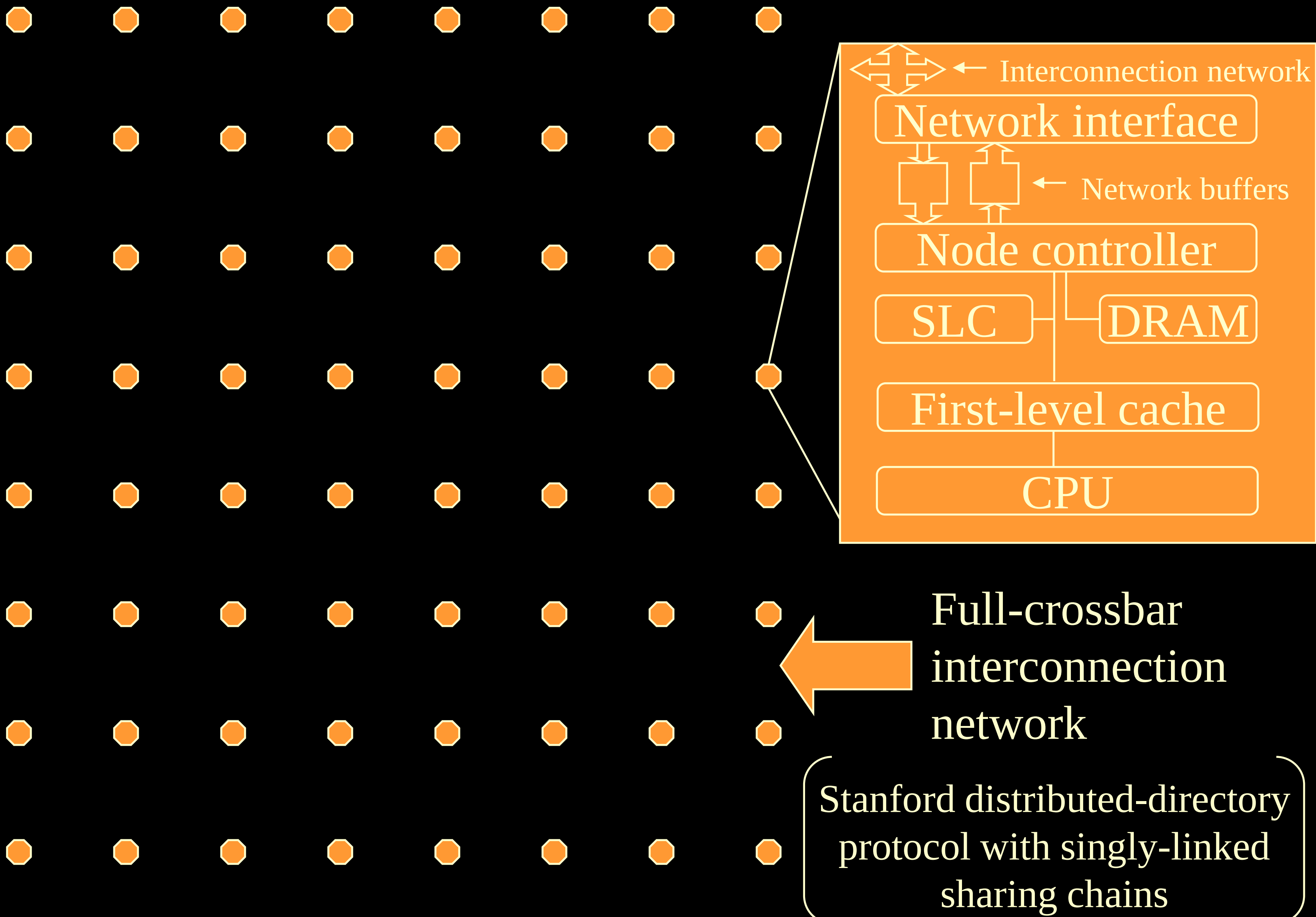
The challenge

- Caching is a great way to enhance *best-case*
- Worst-case is terrible
- What powerful ideas do we have for dealing with worst-case contention?
  - Combining
  - Randomisation
- How can we use caching most of the time, while using random data routing/placement and combining to avoid worst-case contention?

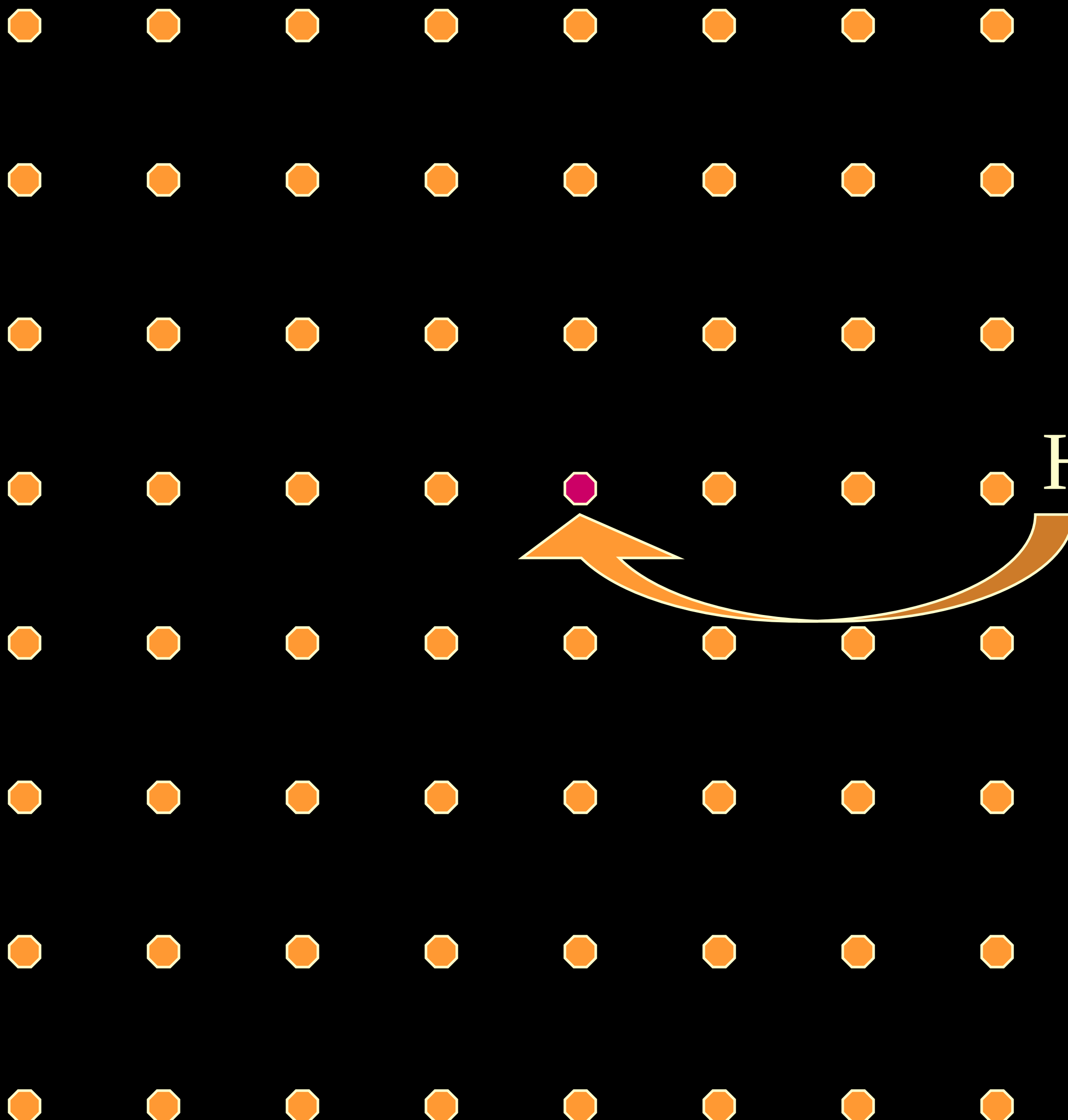
This talk

- Introduce proxying concept
- Briefly review results presented in earlier papers
- Introduce adaptive proxying
- Present simulated benchmark results
- Show that adaptive, reactive proxying can improve performance of susceptible applications
- With no reduction in performance of other applications

Context: large CC-NUMA multicomputer



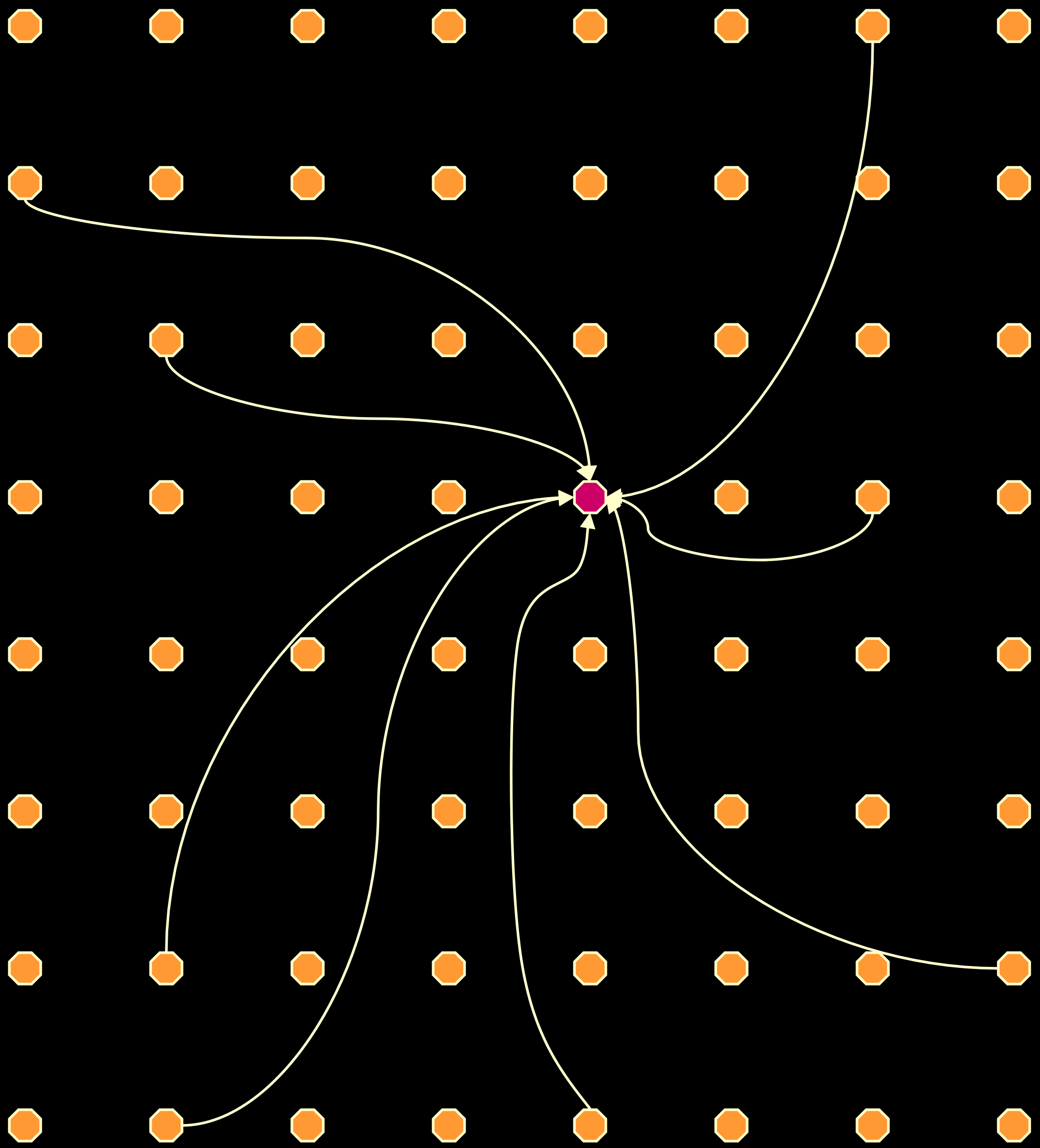
The home node for a memory address



Home for location x

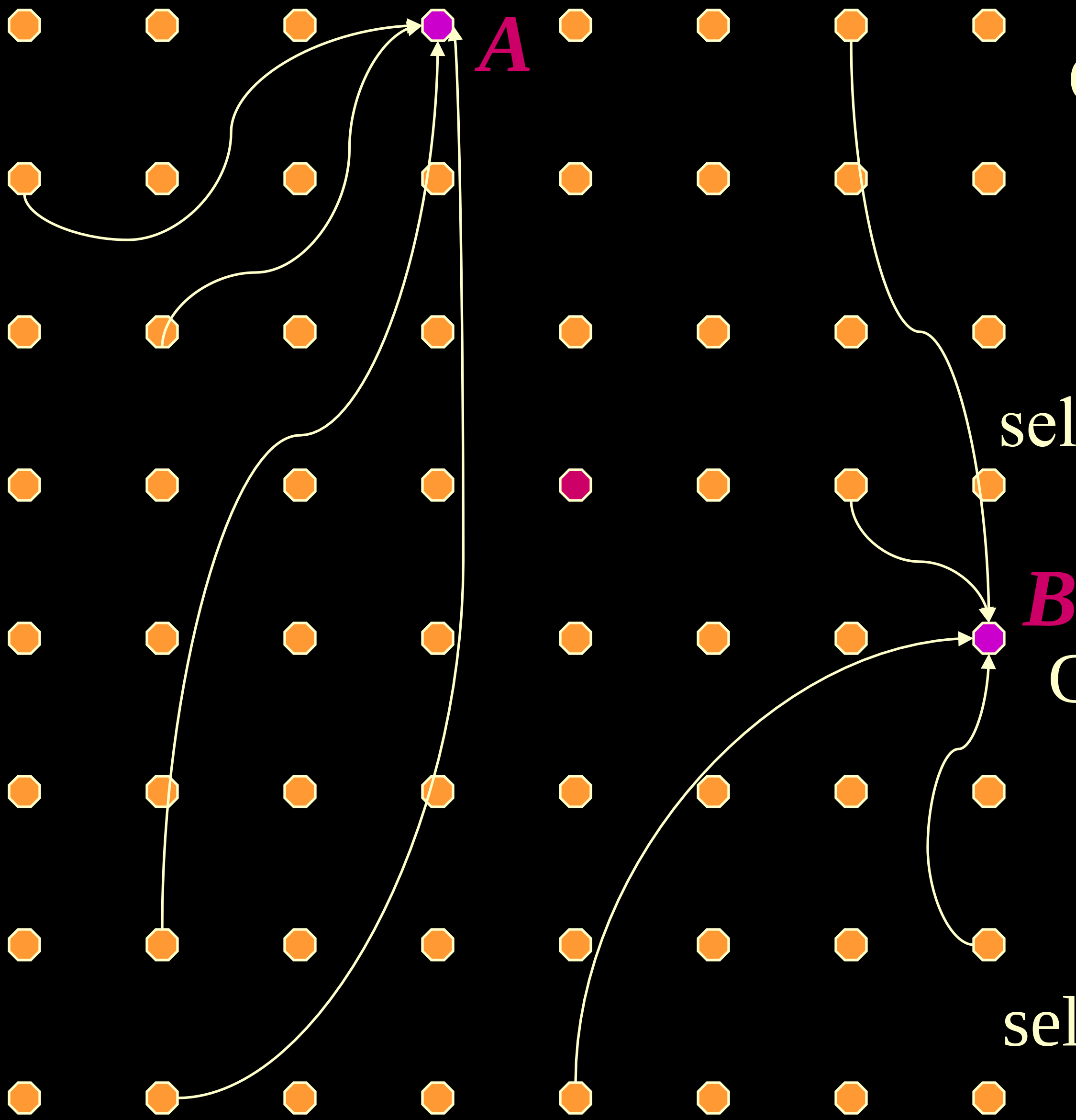
Home is determined by *high-order* bits of *physical* address. All cache misses go first to home to locate a CPU holding a valid copy

Widely-read data



Suppose many CPUs have read misses on the same address at the same time

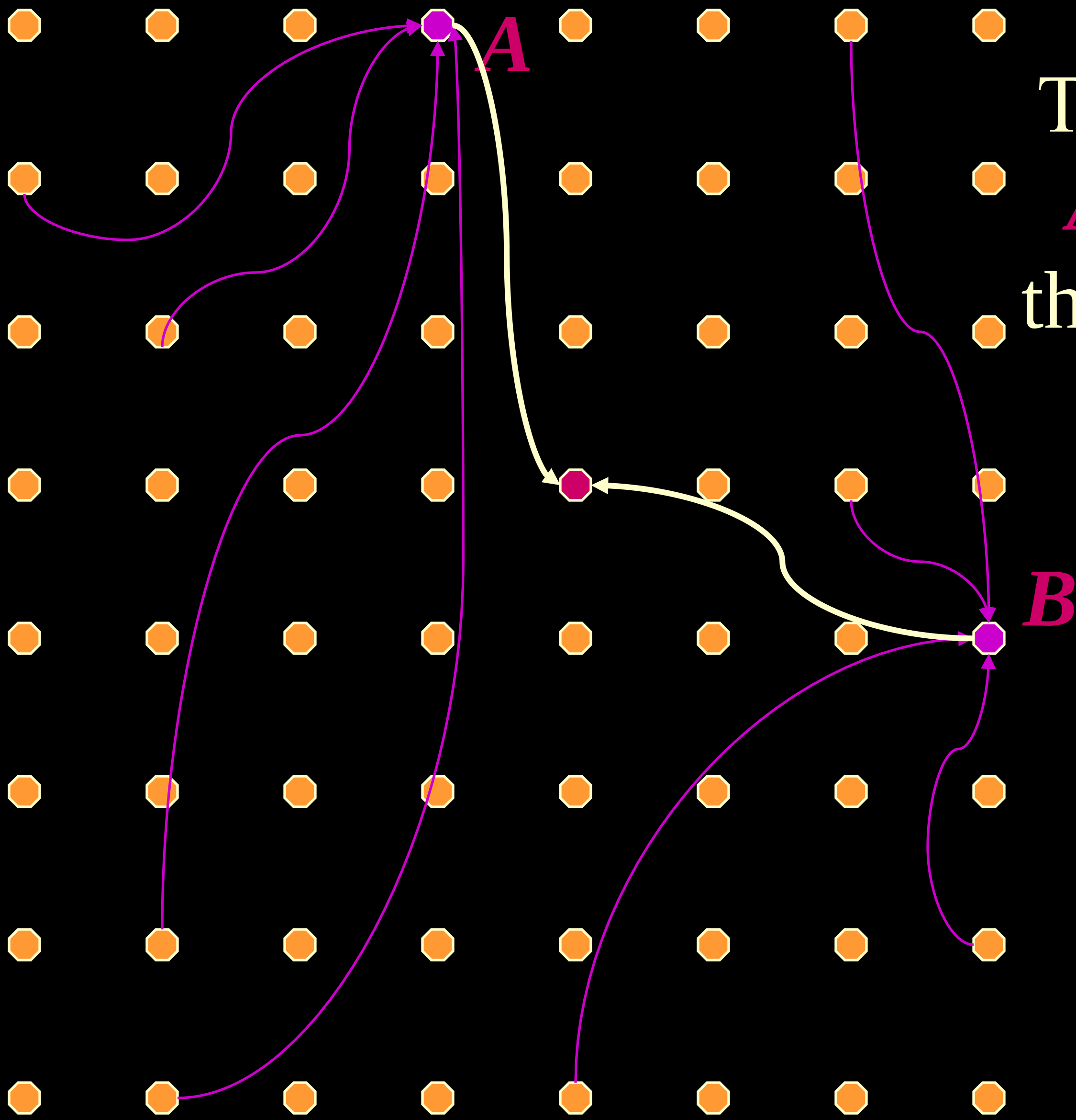
The basic idea: proxies



CPUs in ***left*** half of machine send their read request to pseudo-randomly selected "proxy" node ***A***

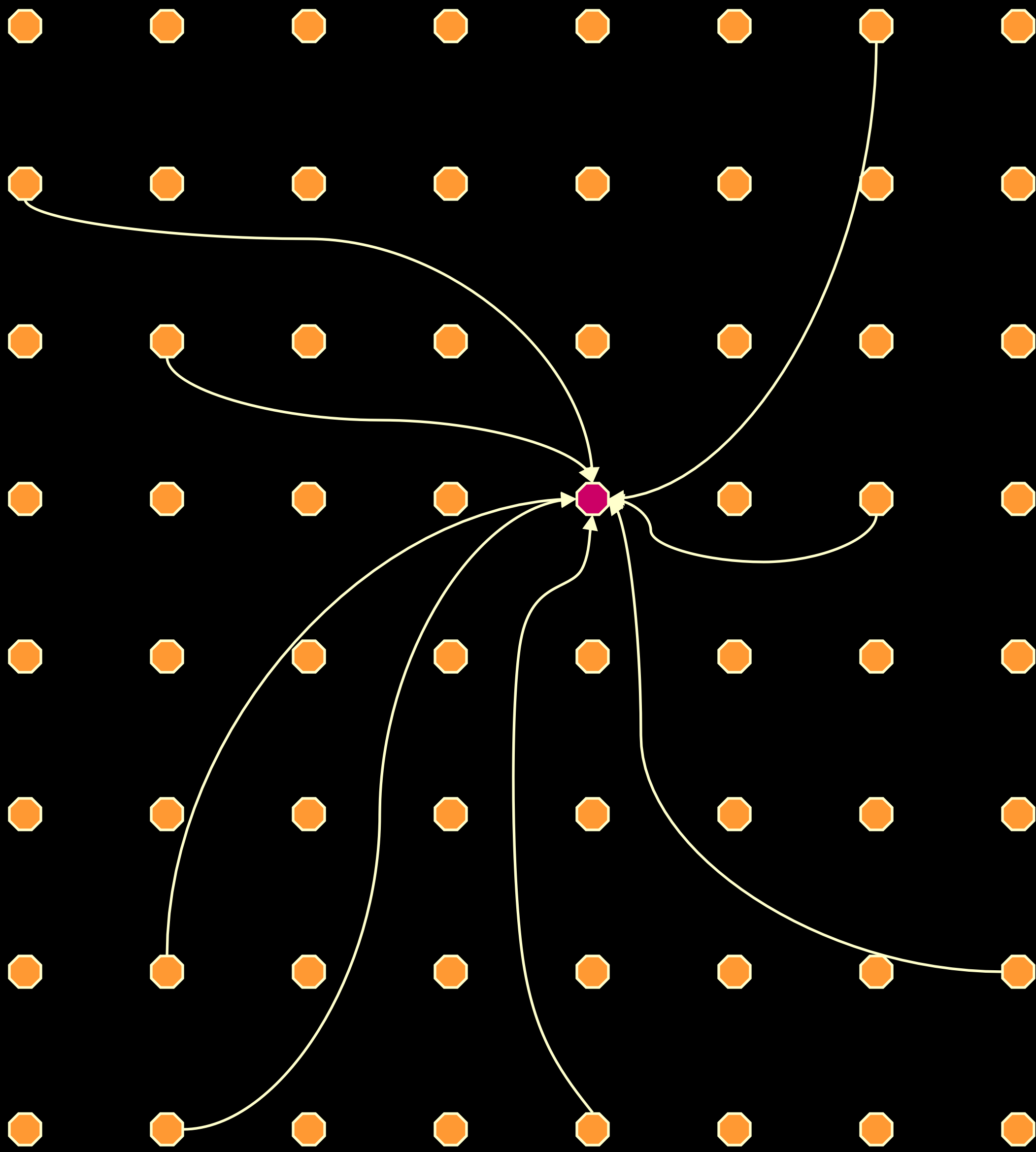
CPUs in ***right*** half of machine send their read request to pseudo-randomly selected "proxy" node ***B***

Proxies - forwarding



The "proxy" nodes ***A*** and ***B*** forward the read requests to the home

Reading the next location...



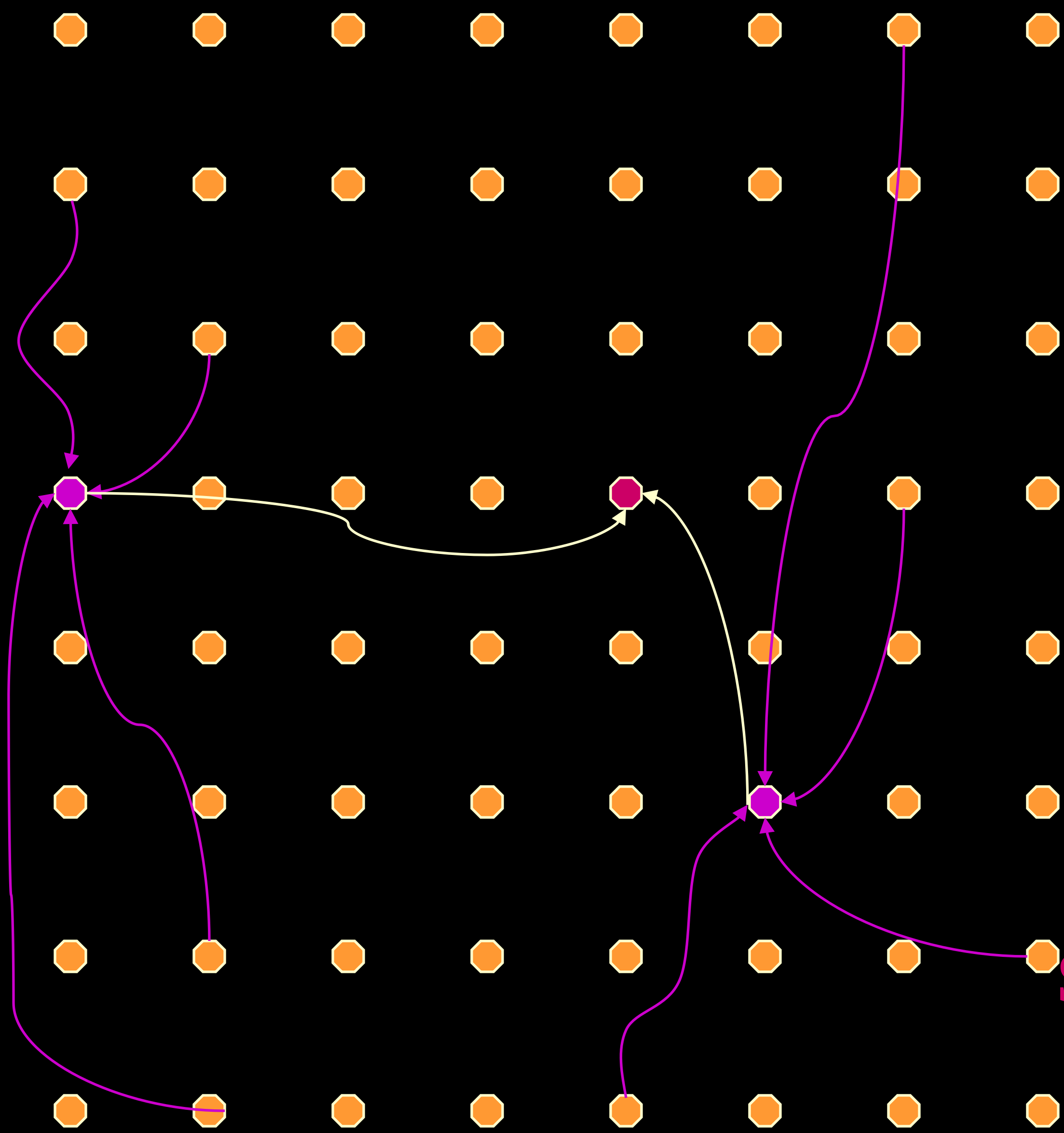
Suppose many CPUs have read misses on the same address at the same time...

and then they all access the next cache line

Locations on same page have same home node

So they all contend again for the same home node

Reading the next location… randomisation



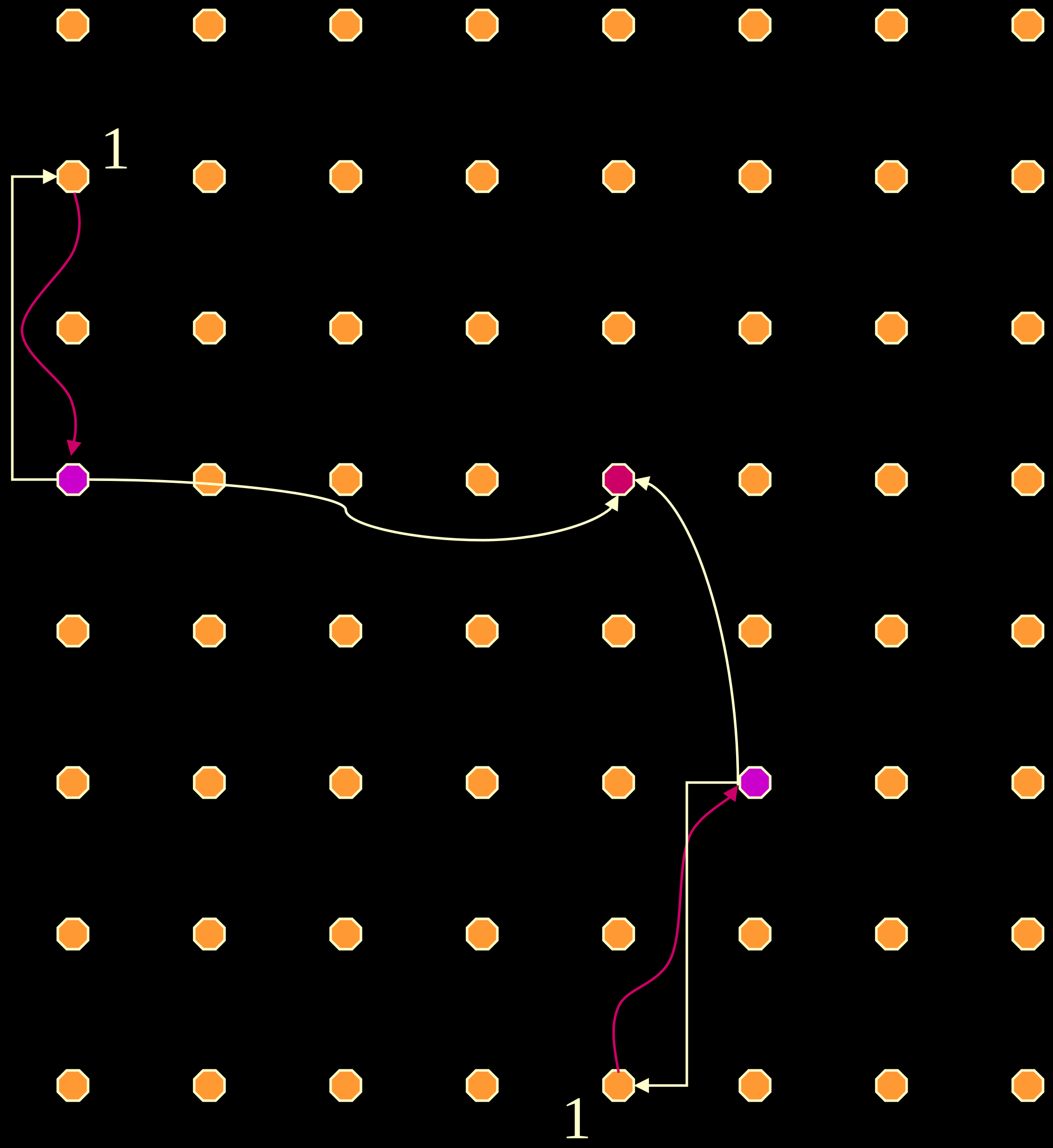
Suppose many CPUs have read misses on the same address at the same time...

and then they all access the next cache line

(Proxy is selected pseudo-randomly based on low-order address bits)

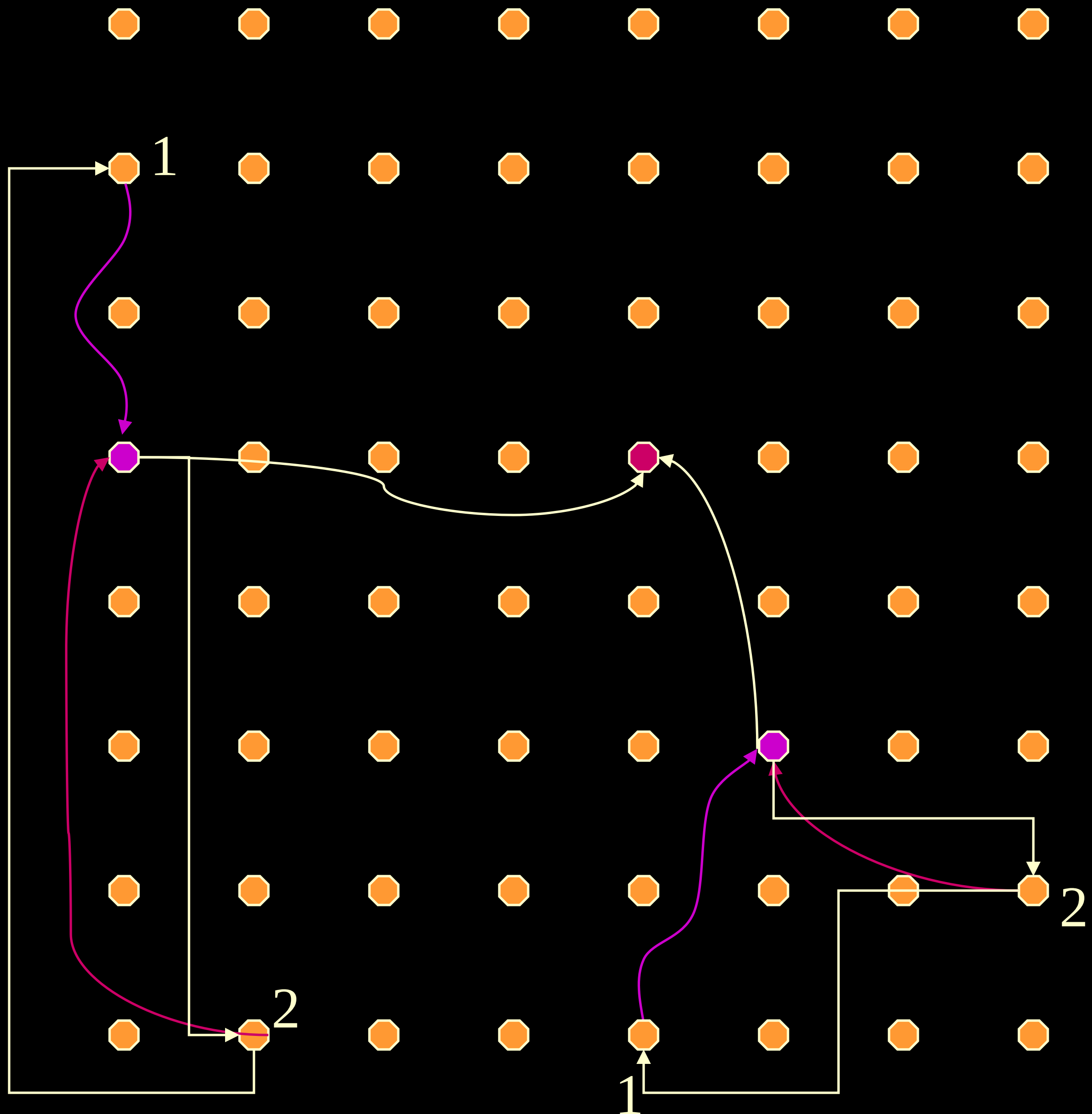
So contention is spread across nodes, read is serviced in parallel

Combining - Pending Proxy Request Chain



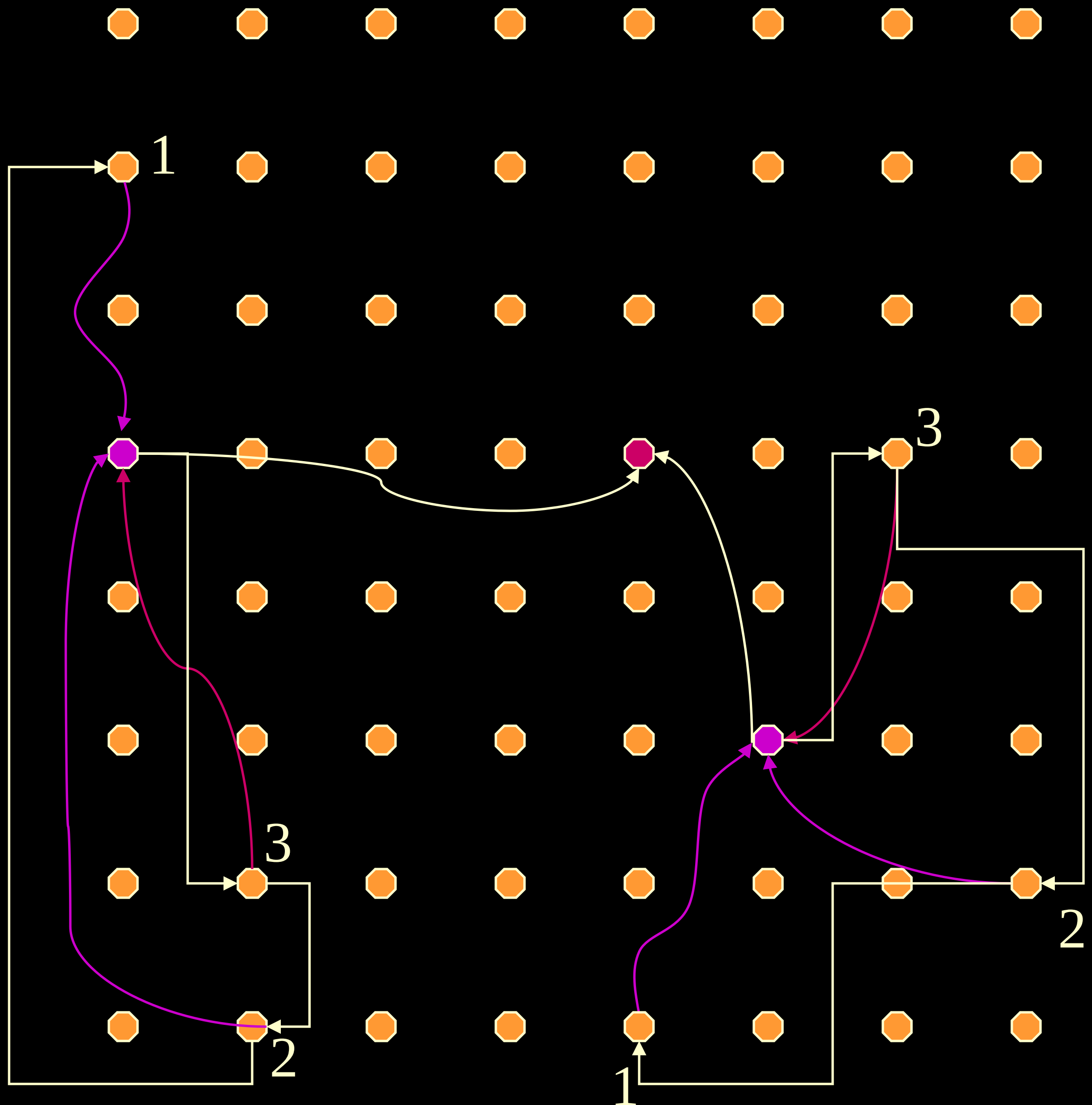
As each read request is received by the proxy, the client as added to a chain of clients to be informed when the reply arrives

Combining - Pending Proxy Request Chain



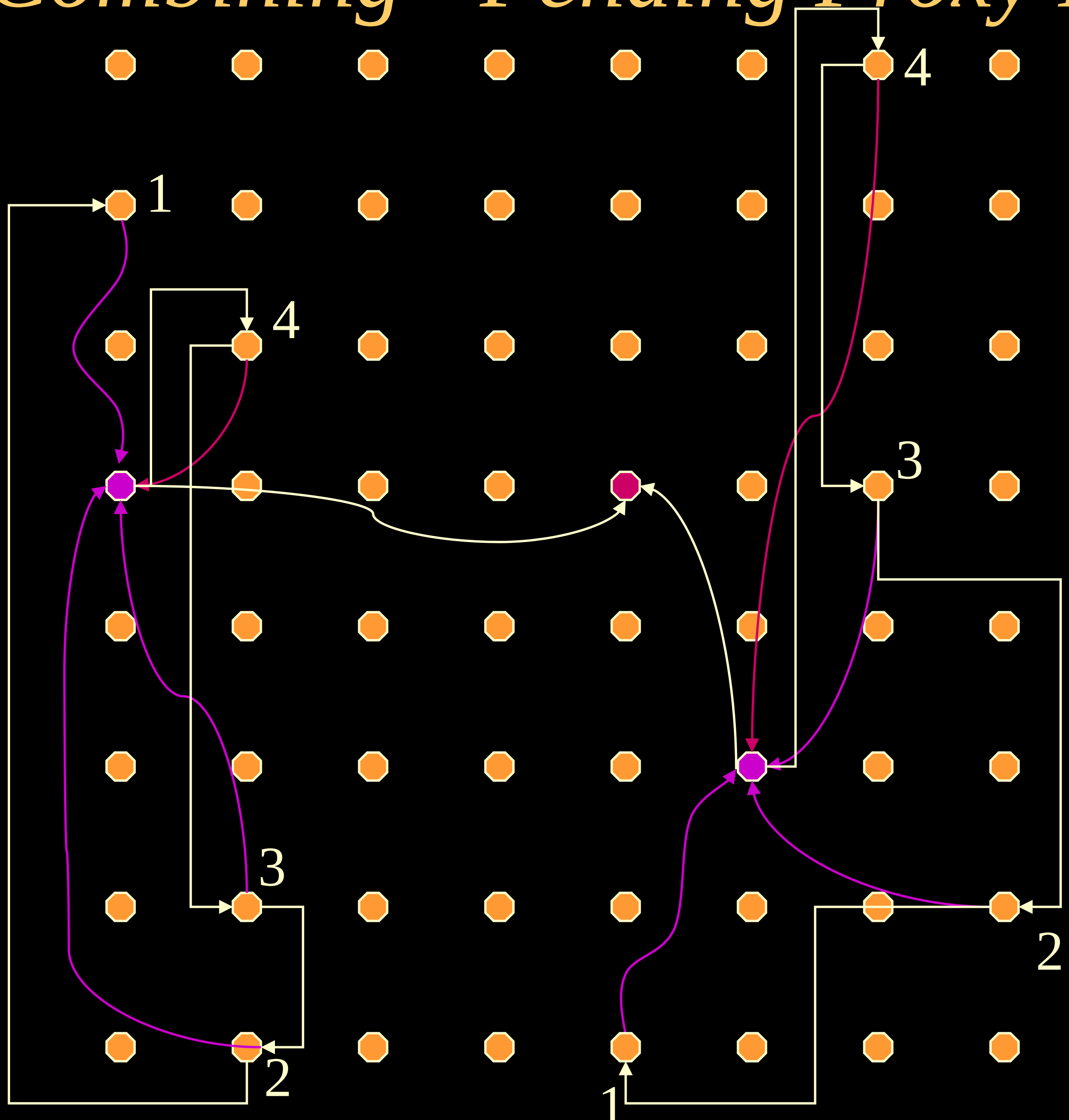
As each read request is received by the proxy, the client as added to a chain of clients to be informed when the reply arrives

Combining - Pending Proxy Request Chain



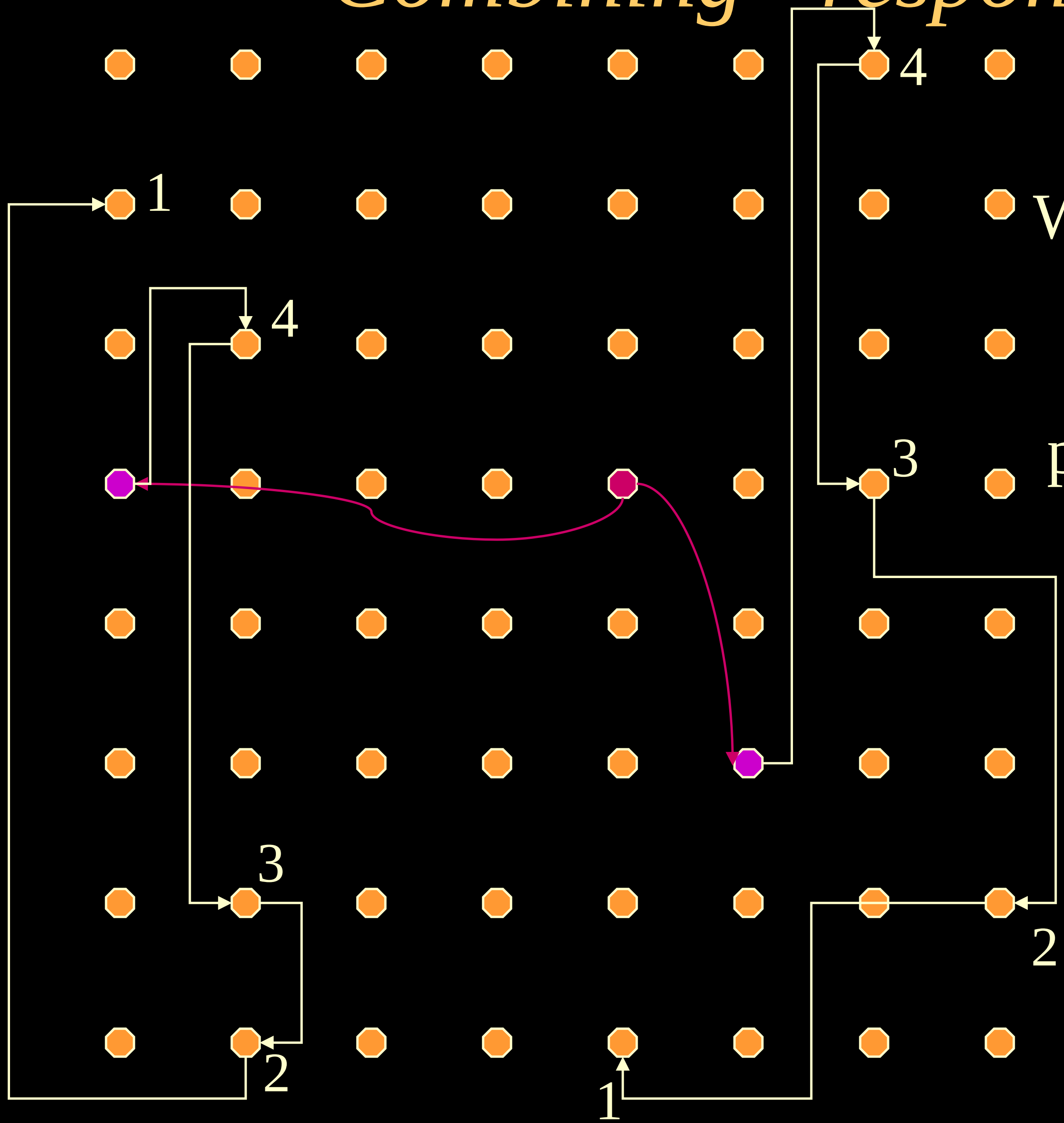
As each read request is received by the proxy, the client as added to a chain of clients to be informed when the reply arrives

Combining - Pending Proxy Request Chain



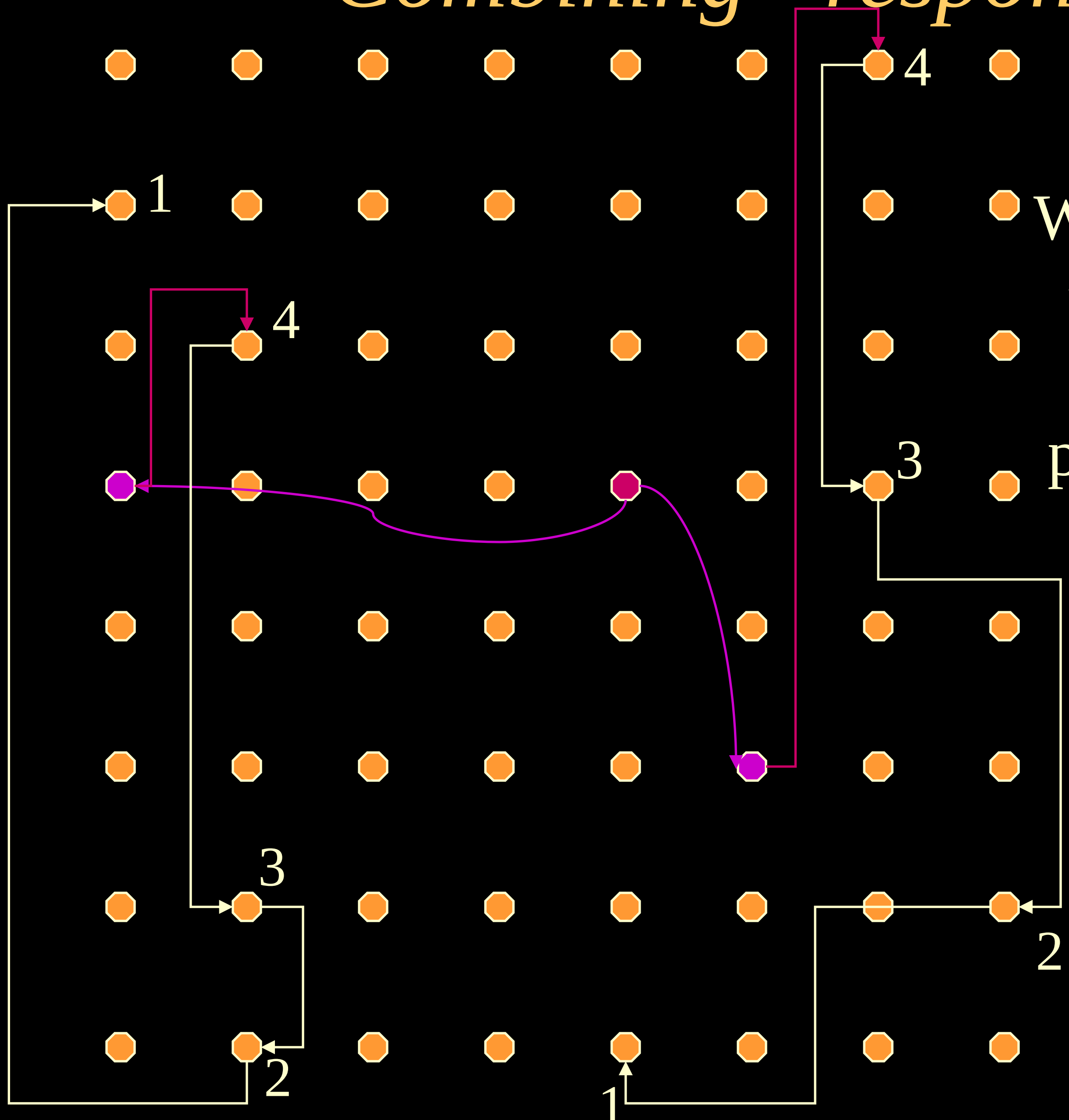
As each read request is received by the proxy, the client as added to a chain of clients to be informed when the reply arrives

Combining - responding to clients



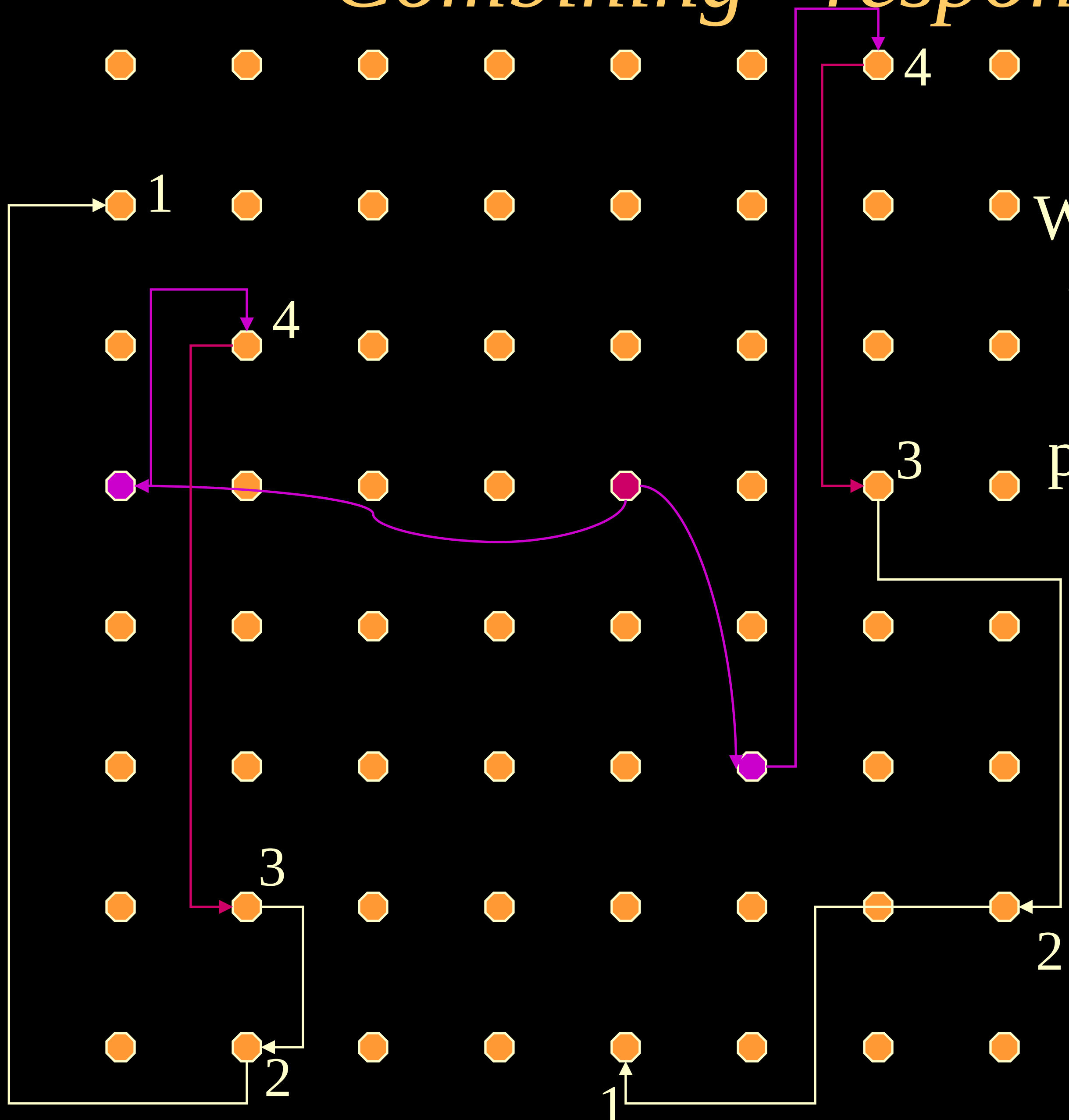
When the reply arrives, the cache line data is forwarded along the proxy pending request chain

Combining - responding to clients



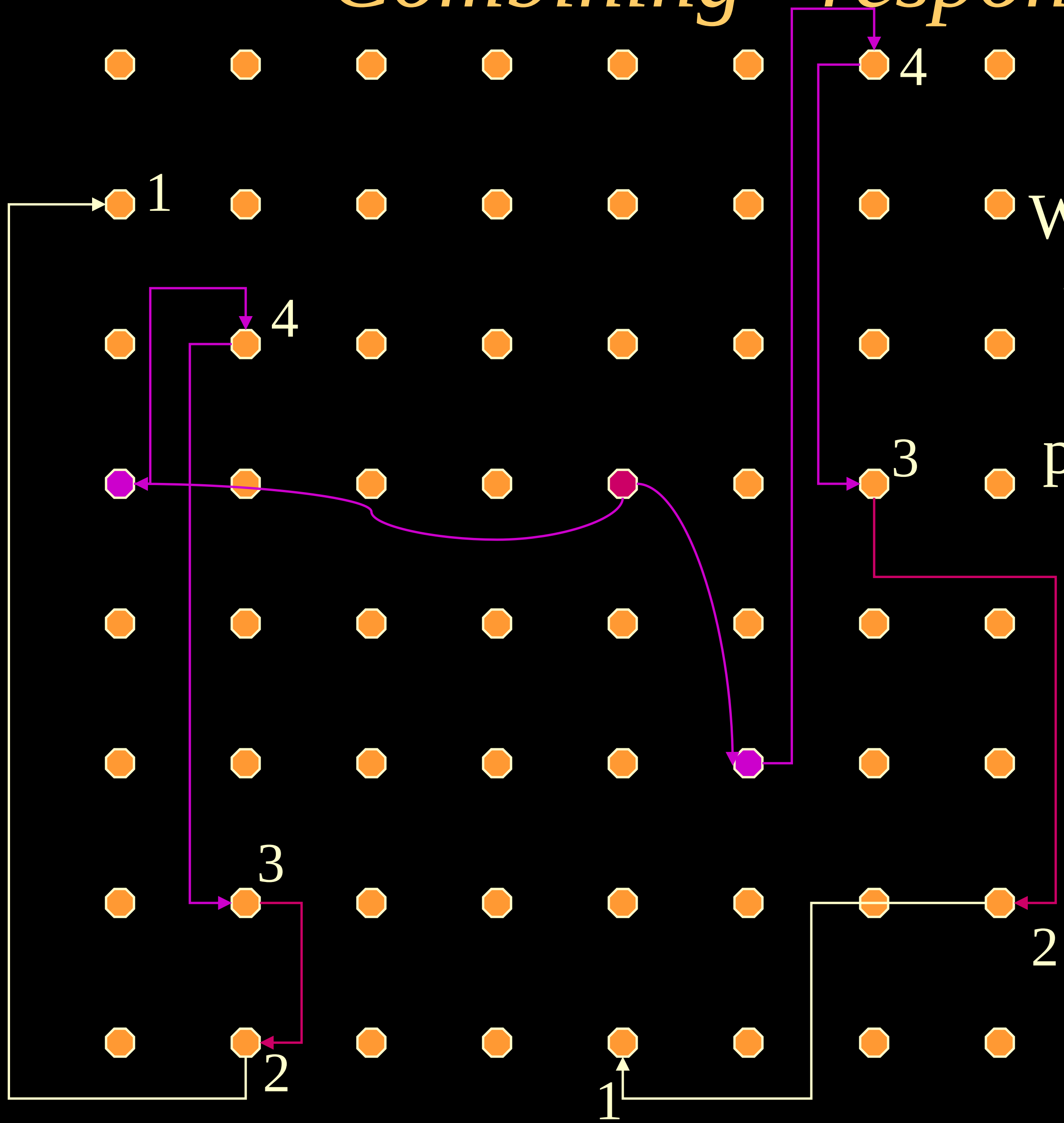
When the reply arrives, the cache line data is forwarded along the proxy pending request chain

Combining - responding to clients



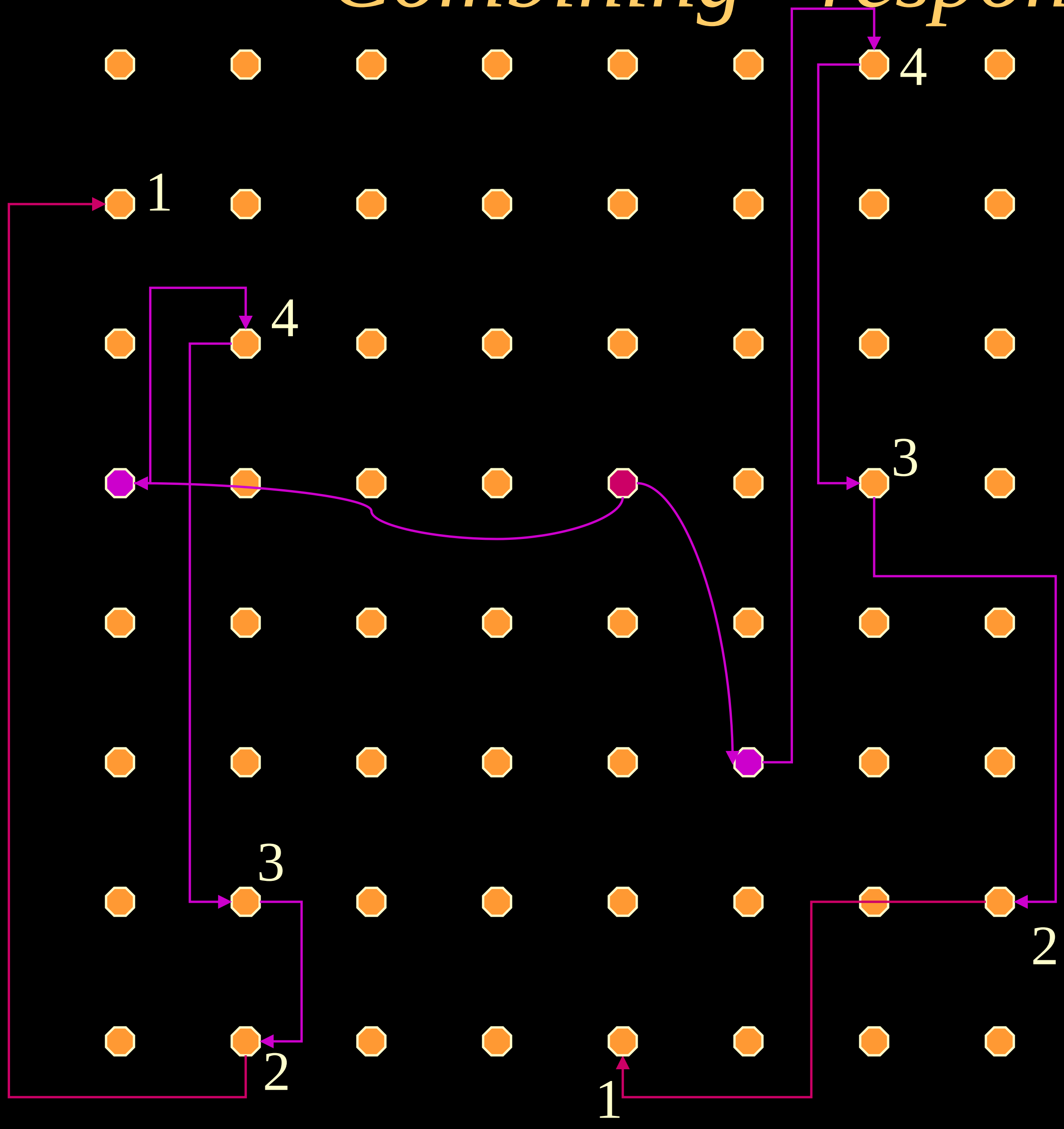
When the reply arrives, the cache line data is forwarded along the proxy pending request chain

Combining - responding to clients



When the reply arrives, the cache line data is forwarded along the proxy pending request chain

Combining - responding to clients



We chose a linked pending chain to minimise space overhead and make proxying easy to add to node controller design

Previously-published results

- (Euro-Par'96) Proxying improves 512x512 Gauss Elimination by >28% on 64 processors
  - But slows most other apps down, so has to be controlled by programmer
- (Euro-Par'98) Reactive proxies: send read request to proxy if request is NAKed due to buffer full
  - Reactive proxying doesn't slow any apps
  - But performance improvement for GE only 21-23%
  - Some other apps show promising 3-10%
- (HPCS'98) With reactive proxies, first-touch page placement is always better than round-robin (and at least as good as without proxying)

This paper

- Adaptivity: if a “recent” request to node ***i*** was NAKed, assume the buffer is still full and route a read request directly to a proxy
- Should proxy retain recently-proxied data?
  - Yes: space is allocated for proxied data in the proxy node’s own SLC, which is kept coherent using the usual protocol
  - No: a separate proxy buffer points to outstanding proxy pending request chains
  - Yes: this separate proxy buffer retains proxied data, which is kept coherent using the usual protocol

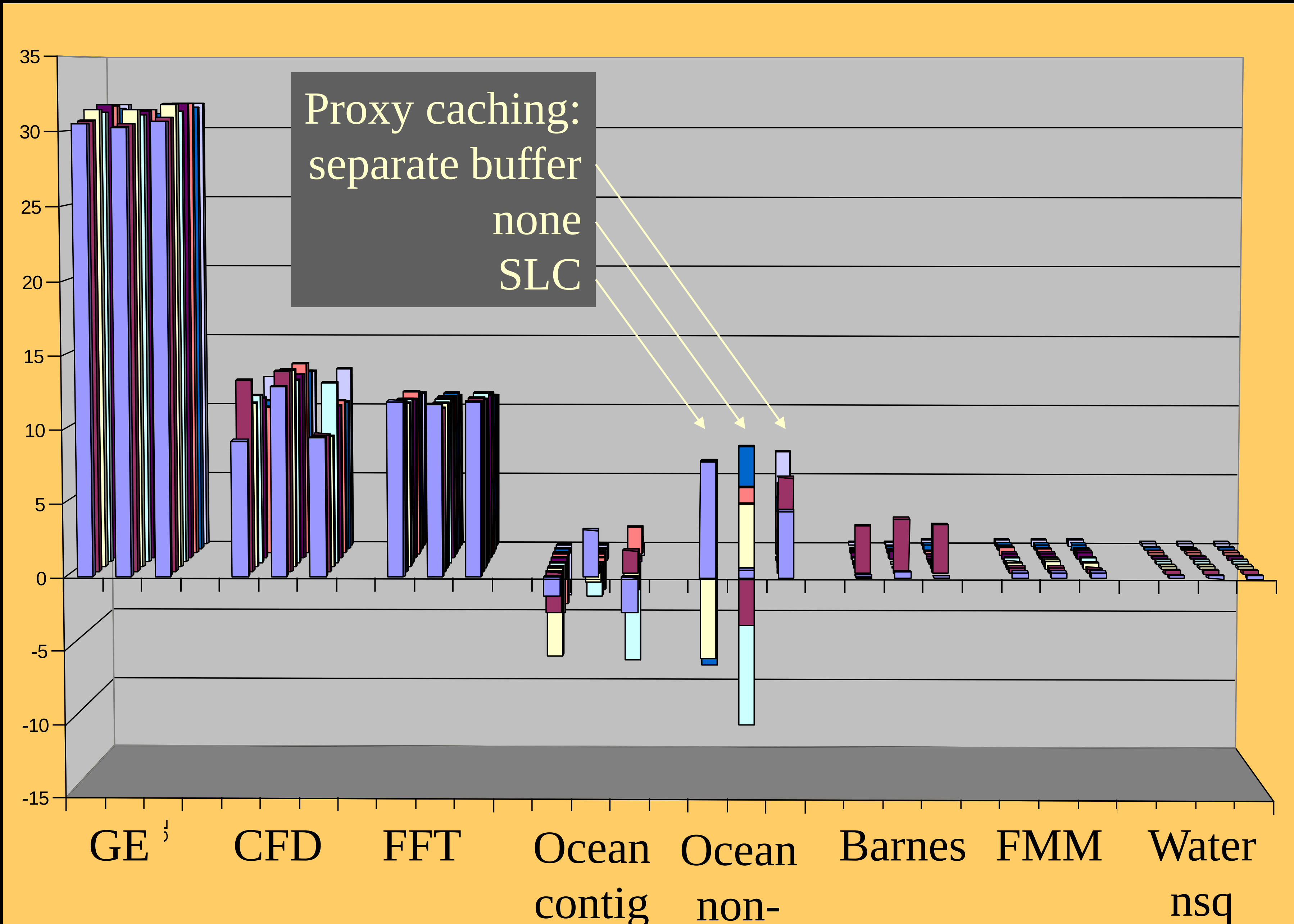
This talk

- Adaptivity: if a “recent” request to node ***i*** was NAKed, assume the buffer is still full and route a read request directly to a proxy

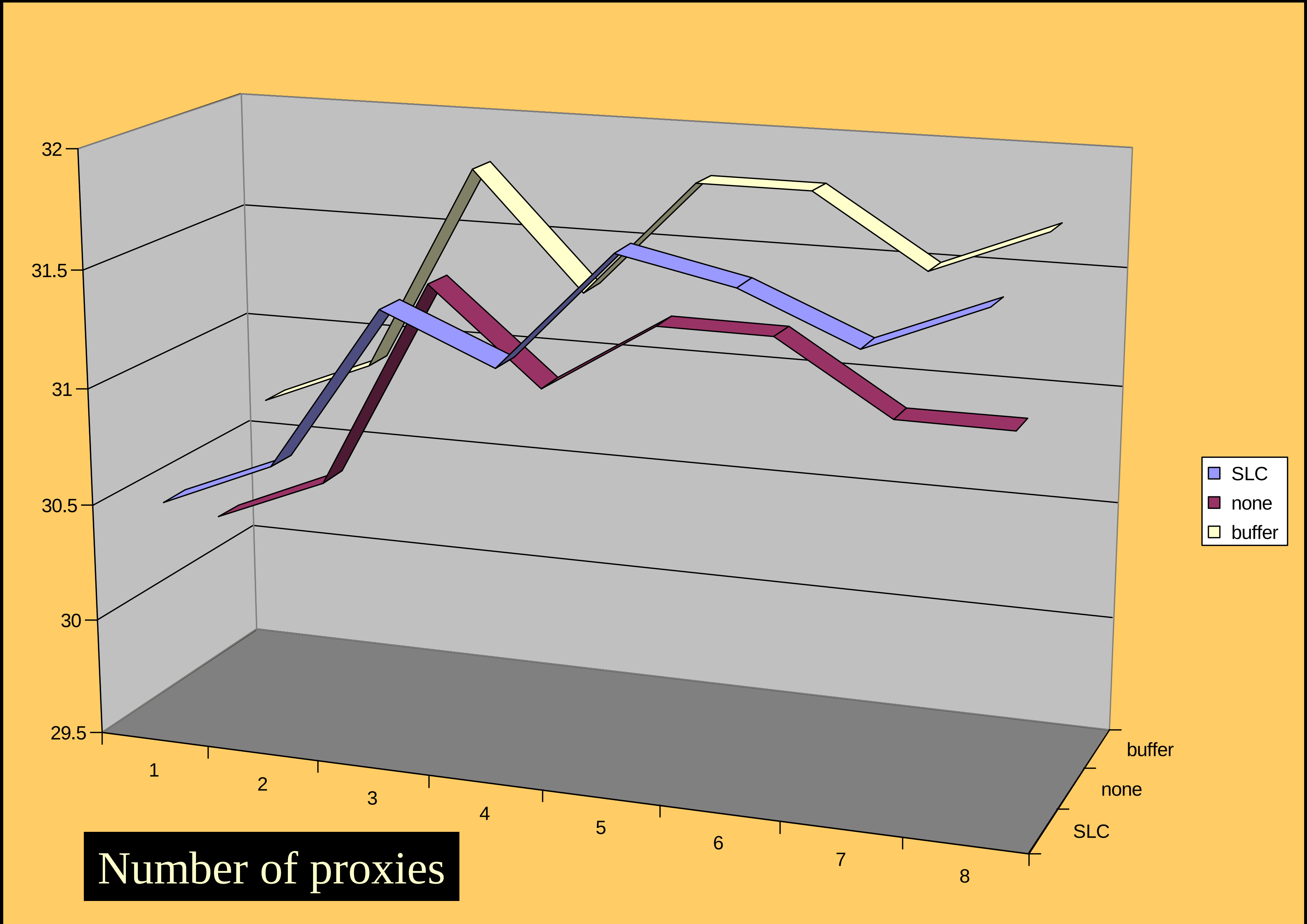
This talk

- Adaptivity: if a "recent" request to node ***i*** was NAKed, assume the buffer is still full and route a read request directly to a proxy
- Each proxy *x* maintains two vectors:
  - LB[*y*]: time when last NAK received by *x* from *y*
  - PP[*y*]: current "proxy period" for reads to node *y*
- When a NAK arrives at *x* from *y*, PP[*y*] is
  - incremented if LB[*y*] is within given window
  - decremented otherwise
  - (unless $PP_{min} \le PP[y] \le PP_{max}$)

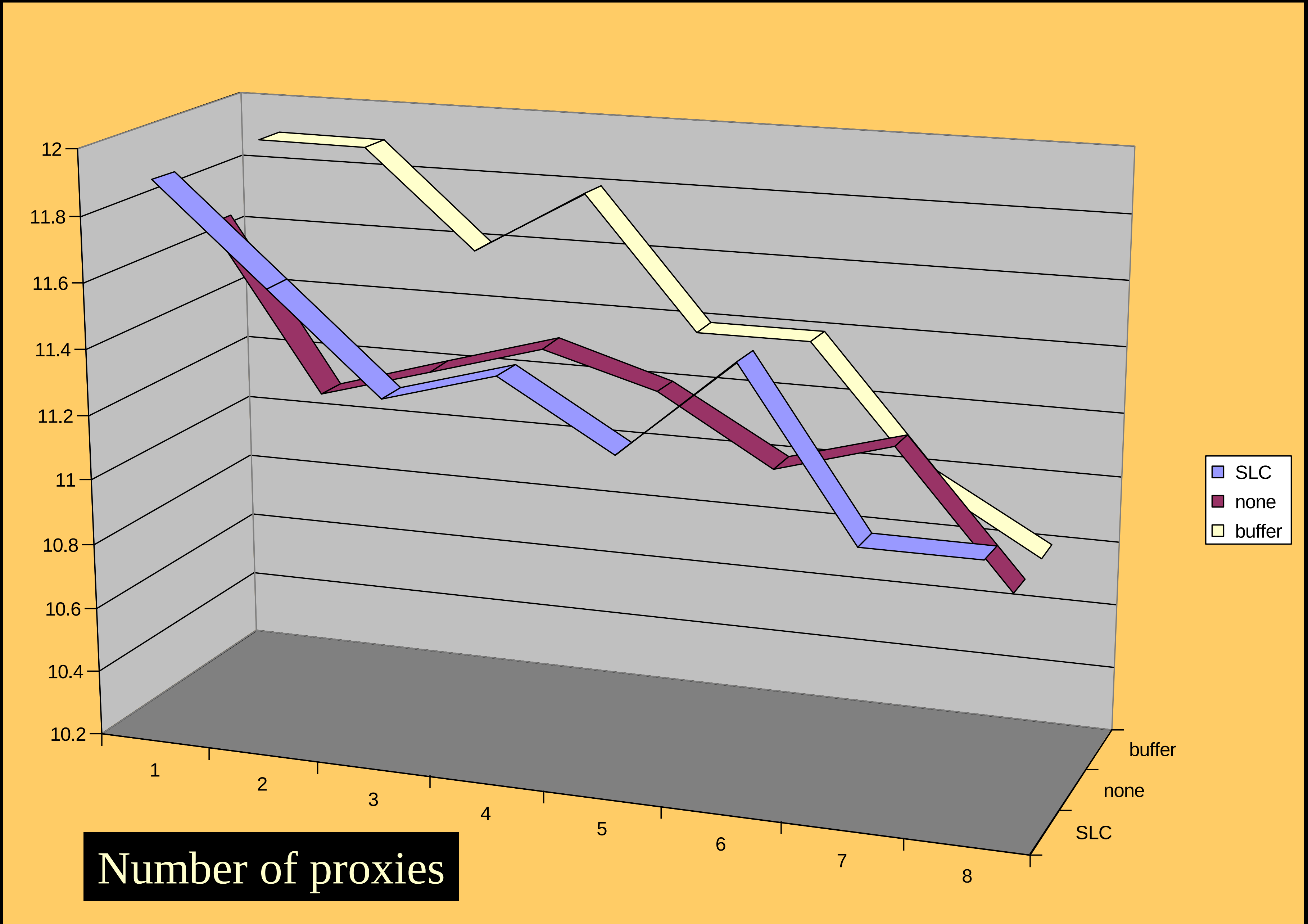
Results - simulated benchmark execution



Gaussian elimination

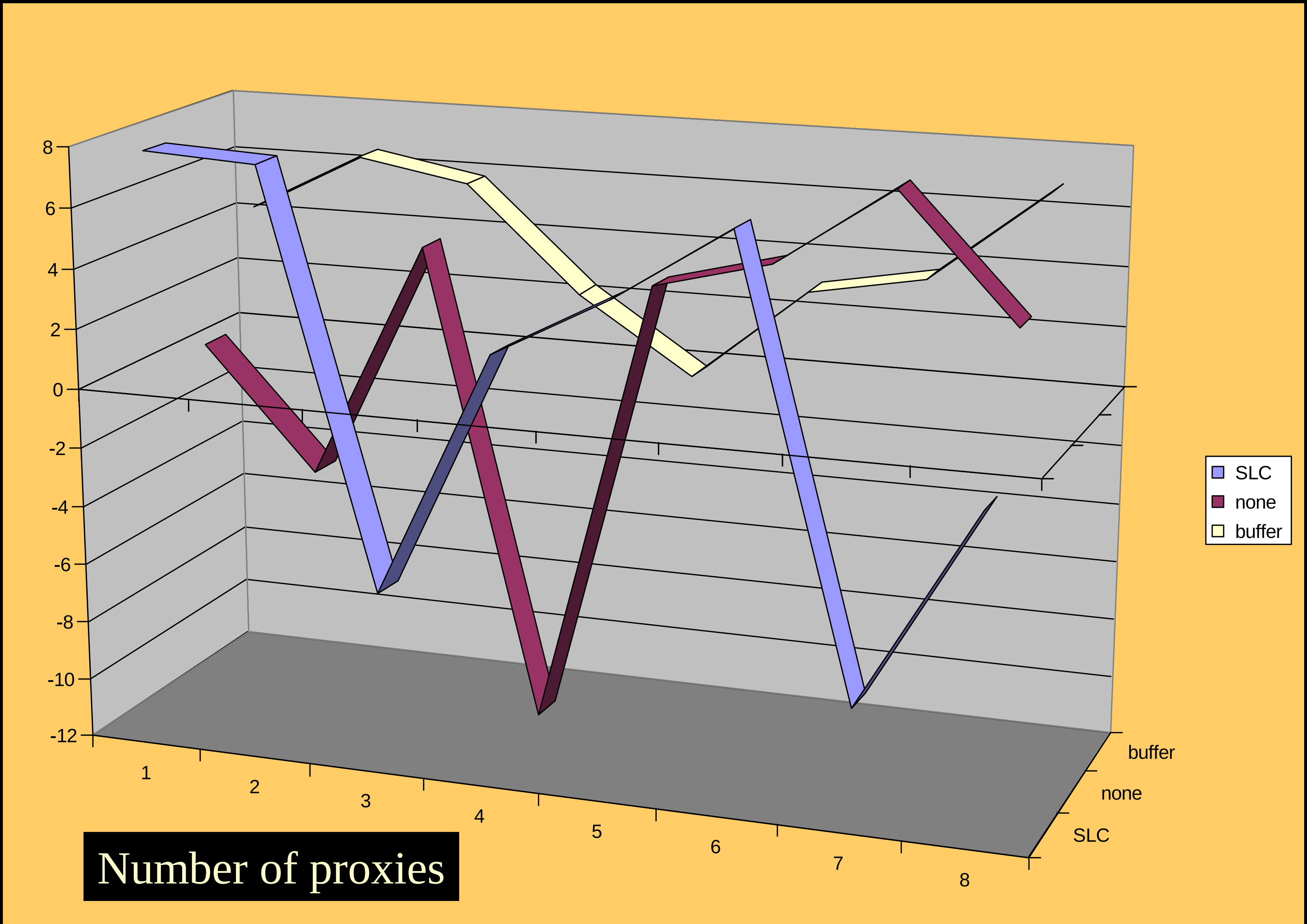


FFT

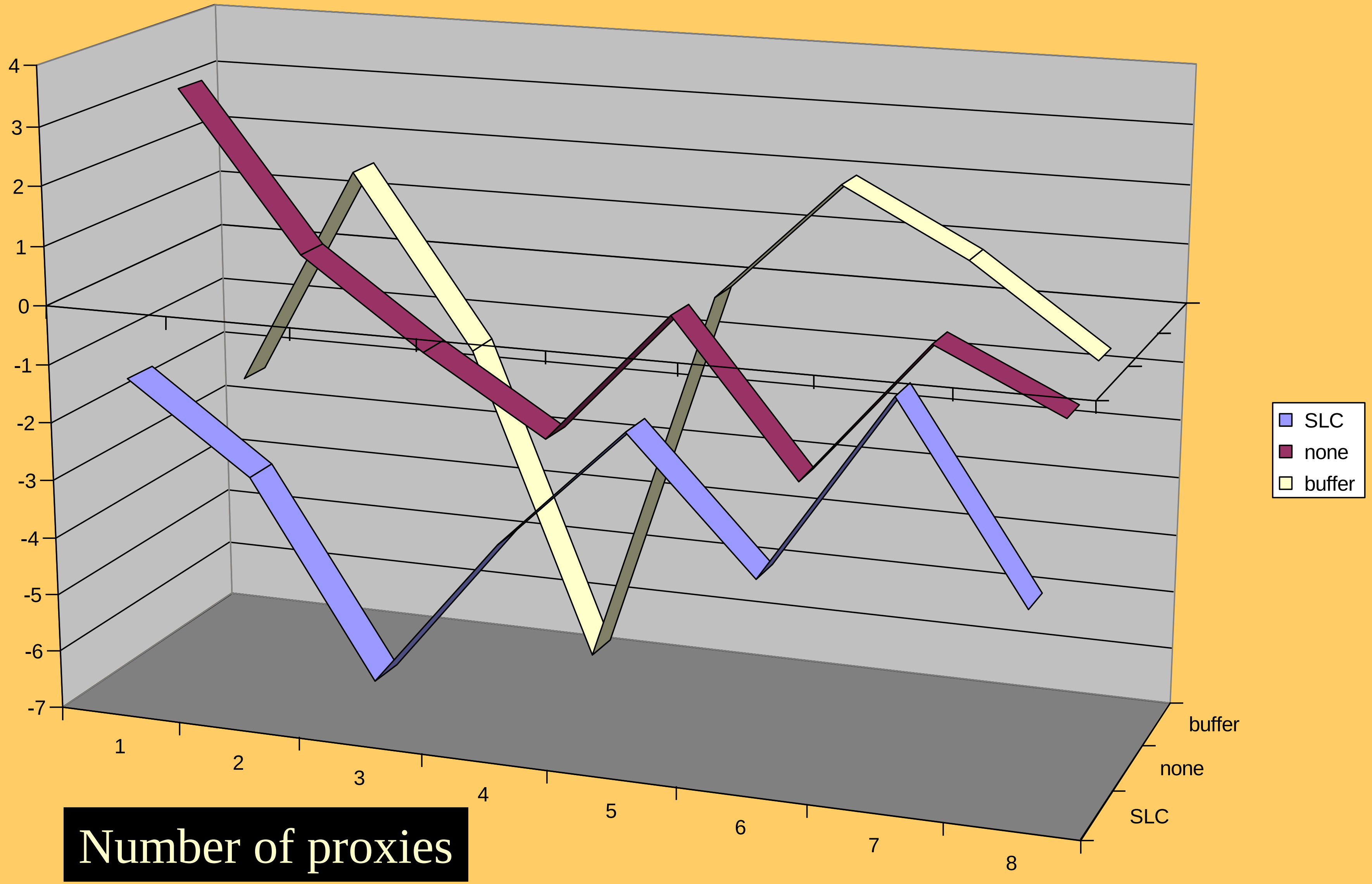


12
11.8
11.6
11.4
11.2
11
10.8
10.6
10.4
10.2
1
2
3
4
5
6
7
8
buffer
none
SLC
SLC
none
buffer
Number of proxies

Ocean, non-contiguous storage layout

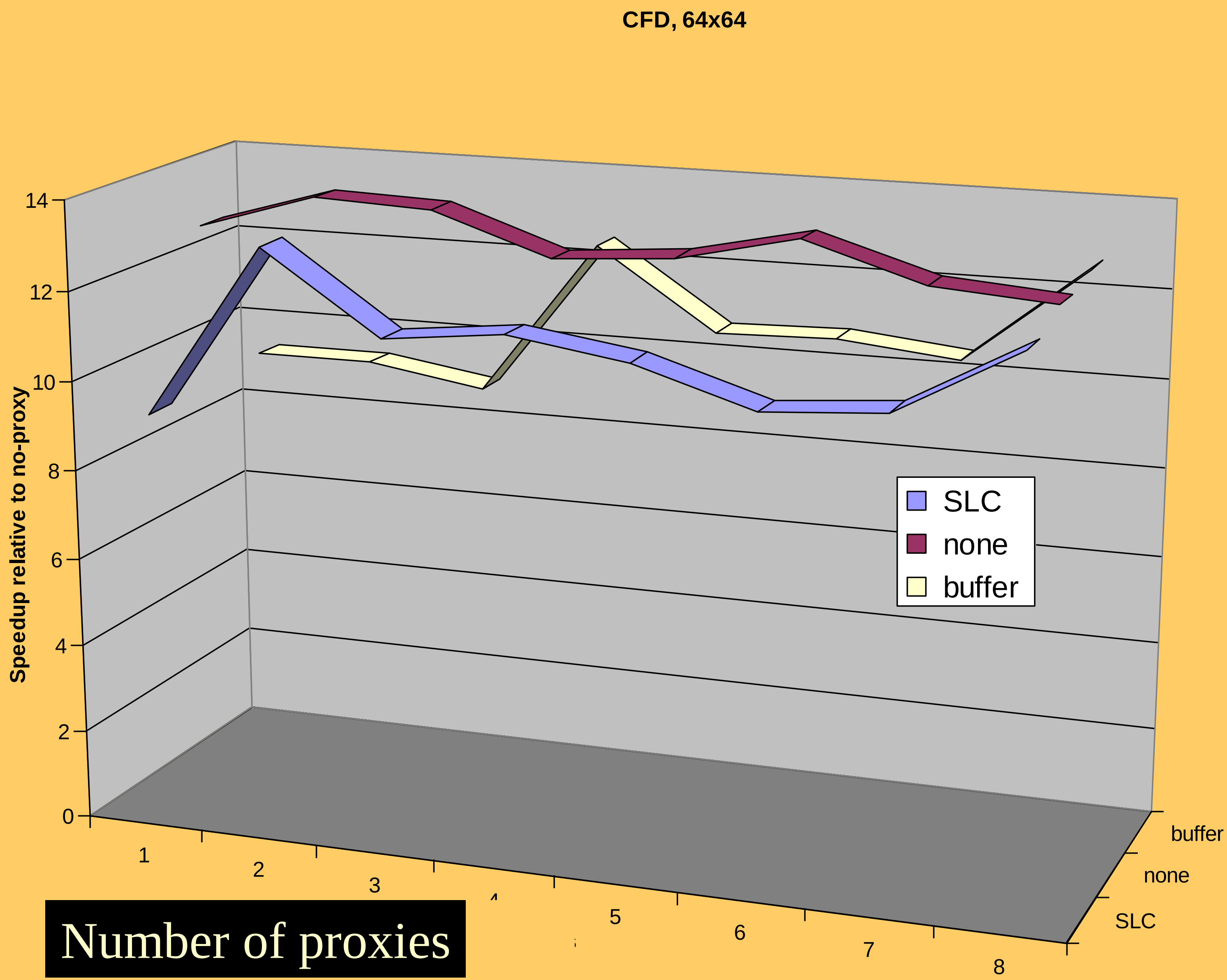


Ocean, contiguous storage layout



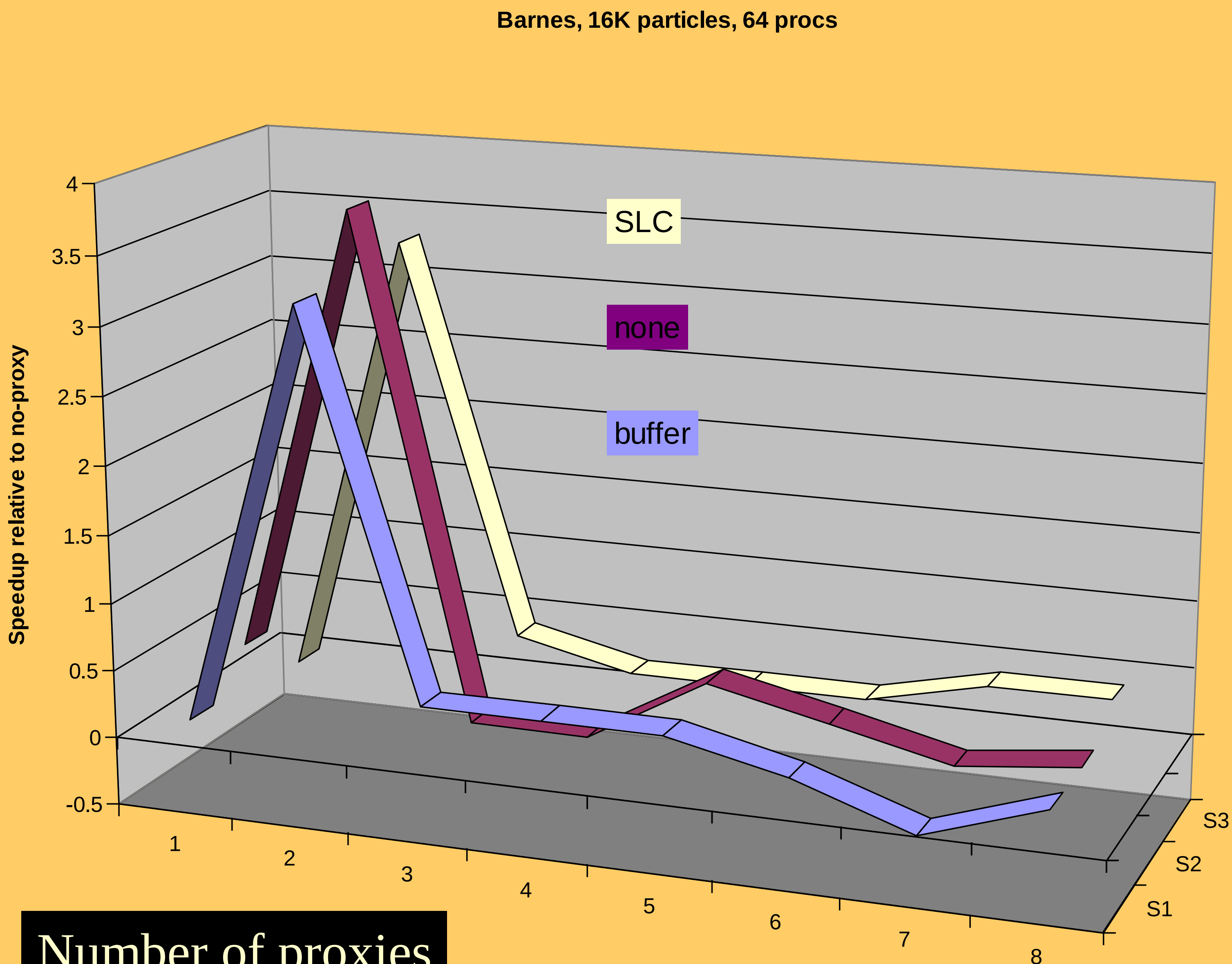
Number of proxies

CFD



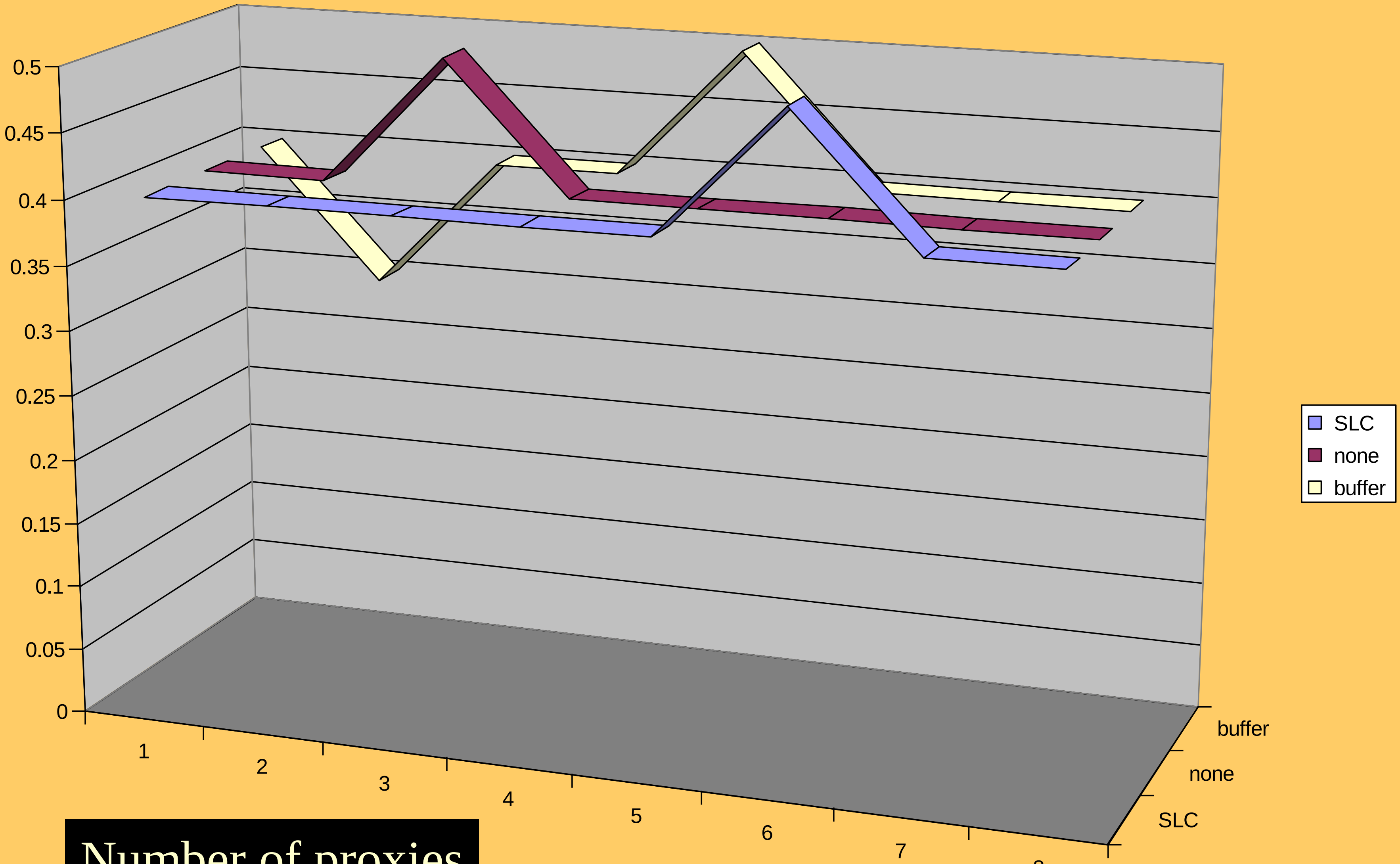
Number of proxies

Barnes-Hut, 16K particles



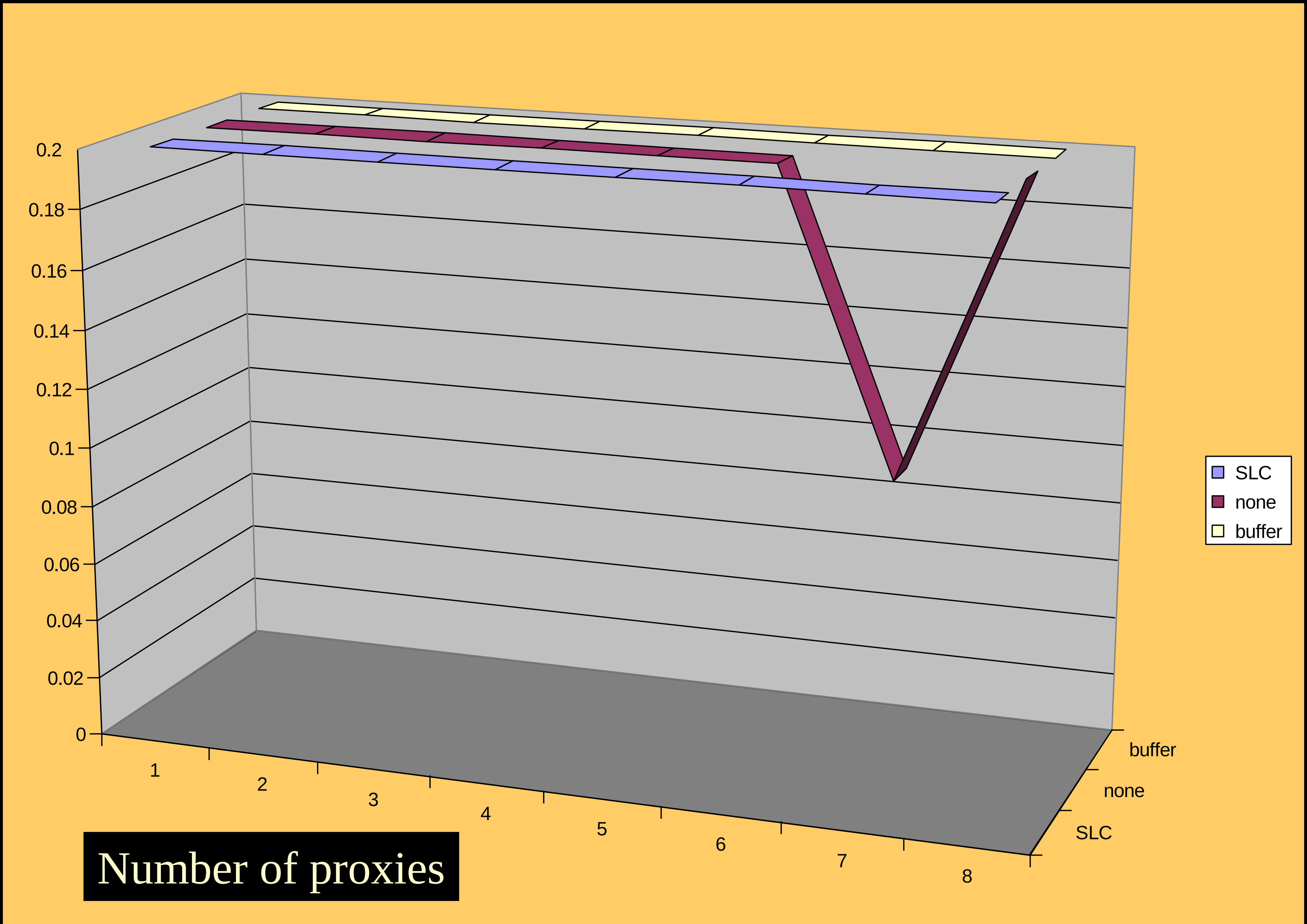
Number of proxies

FMM



Number of proxies

Water-nsq


0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1
2
3
4
5
6
7
8
buffer
none
SLC
SLC
none
buffer
Number of proxies

Conclusions

- There is a lot of scope for further work -
  - more applications, range of architectural parameters
  - clustered interconnection network
  - worst-case performance of writes?
- Proxying can solve serious performance anomalies
  - Using a separate proxy buffer is best
  - Proxying without allocating cache space for the proxied data works remarkably well
  - The optimum number of proxies varies erratically
  - But a conservatively small number of proxies (1 or 2 for 64 procs) is a good choice

Acknowledgements

- Colleagues Tony Field, Andrew Bennett (now with Micromuse), Ashley Saulsbury (now with Sun Labs)
- Funding: EPSRC PhD Studentship and Research Grant “CRAMP: Combining, Randomisation and Mixed-Policy Caching”