# Data Cache Design and Evaluation for SMT Processors

Ron Y. Pinter
Haggai Yedidya

Dept. of Computer Science
Technion

November 11, 2003

# What is SMT?

- SMT = Simultaneous Multi Threading
- (Relatively) simple and natural enhancement to super-scalar processors
- Several HW threads issue instructions to the (modified) super-scalar pipeline
- Commercially available: Intel’s Xeon processor, with HyperThreading Technology (2 threads, 5% area cost)

# SMT – Why?

- Increase throughput by exploiting thread level parallelism (TLP) as well as ILP
  - Increase the sustained utilization of existing processor resources.
  - Enable the addition and use of extra processor resources (previously limited by ILP)

# SMT – Memory Problem

- Memory access is as slow as before:
  - 1st level cache performance remain a key factor
  - Small code sections can access large amounts of data (loops, fit in instruction cache)
  - Data access is a bigger problem than instruction fetch

  ➢ The bottle gets bigger but the bottleneck does not

- Possible solution: larger caches
  - Larger caches are slower and/or consume more energy
- Another approach: better suited behavior
  - Caching approach originally based on access locality of serial processors

# Research Goals

- What are the characteristics of a good data cache for SMT processors?
  - Use cache building blocks and sizes common to present day data caches
  - Amenable to high performance implementation
  - Measure best configurations and tradeoff relationships

- Does the non-serial case exhibit data access locality?
  - If not:
    - How much disruption does the new access pattern introduce?
    - How can it be dealt with?

# Methodological Framework

- Empirical evaluation, using:
  - A simulator
    - Detailed micro-architectural simulator
    - Only 1st level data cache changed in processor model
  - Several test workloads
  - Key factors of the cache configuration, modeled into the simulator:

| Structure | Logic |
| --- | --- |
| Total size | Hash Function |
| Line size | Replacement Policy |
| Level of associativity | |

# Workloads

- Multi-threaded load
  - Single program and memory space.
  - Multiple threads run to complete program (parameter)
    - Kernel: implementation of a single algorithm kernel
    - Application: real application, heterogeneous behavior

- Multi-process load
  - Each thread is an independent serial program
  - Separate memory spaces
  - Each run is a recipe of several threads

- Parallelism
  - Number of threads represents different work profile
  - Each workload type is tested with different levels of parallelism

# Implementation Environment

- Simulator
  - SMTSIM, written by Dean Tullsen (Univ. of Washington, now at UCSD)
  - DEC Alpha based instruction set
  - User Level Simulator
  - Extended and modified for our purposes

# Multi-Threaded Load

- Parts of the Stanford SPLASH-II suite.
  - Kernels: Cholesky, FFT, Radix
  - Application: Raytrace
- Available as source files
  - Extensive adaptation required
- Problem sizes and measuring run times determined for simulations
- 1,2,4,8 threads were used, using same problem sizes

# Multi-Process Load

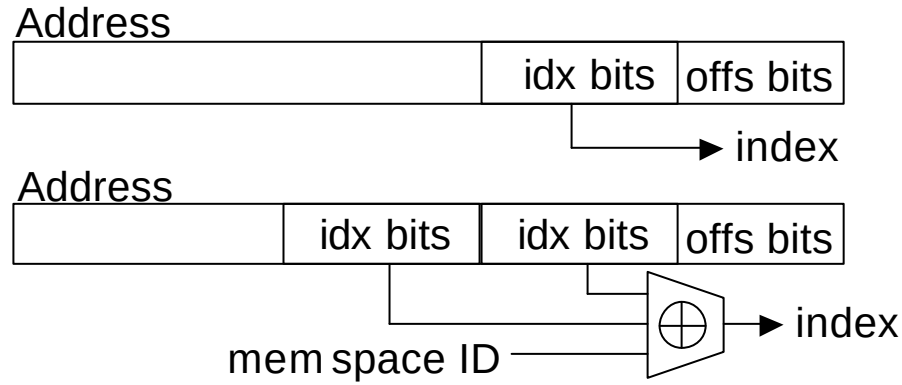
- Parts of the SPEC CPU2000 suite
  - Benchmarks: Equake, Crafty, Mgrid, Gcc, Gzip
  - 12 overall running configurations
- Downloaded from Dean Tullsen's SMTSIM homepage
  - Ready to run- no adaptation needed
- Commonly accepted benchmarks
  - Problem sizes and measuring run times predetermined
- Recipes conjured for 2,4,8 threads in 2 combinations
  - All benchmarks run as single thread for reference

# Cache Key Factors: Structure

- Cache structure:
  - Size/line size:
    - 16KB, 64B/line
    - 32KB, 64B/line
    - 16KB, 32B/line
    - Helps differentiating the effects of size, line size and total number of lines
  - Level of associativity: 1,2,4,8,16,32,64
    - Enables measuring the tradeoff between number of associative sets and their size

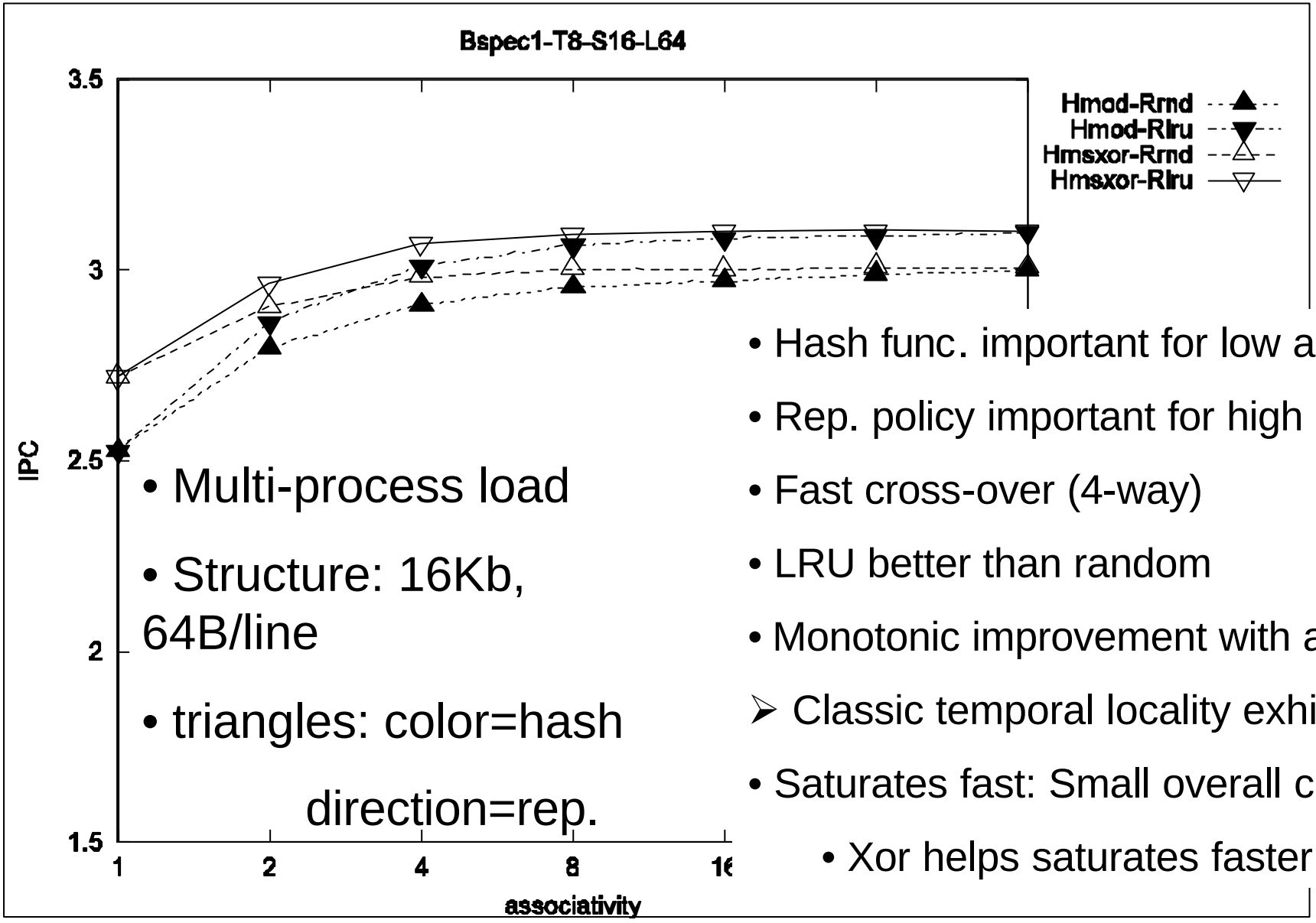
# Cache Key Factors: Logic

- Cache logic:
  - Hash functions:
    - Modulo
    - Xor




  - Replacement policies:
    - LRU
    - Random: each associative set has a wrap-around counter, incremented on every access (R/W).

# Results: Multi-Process, 8 Threads



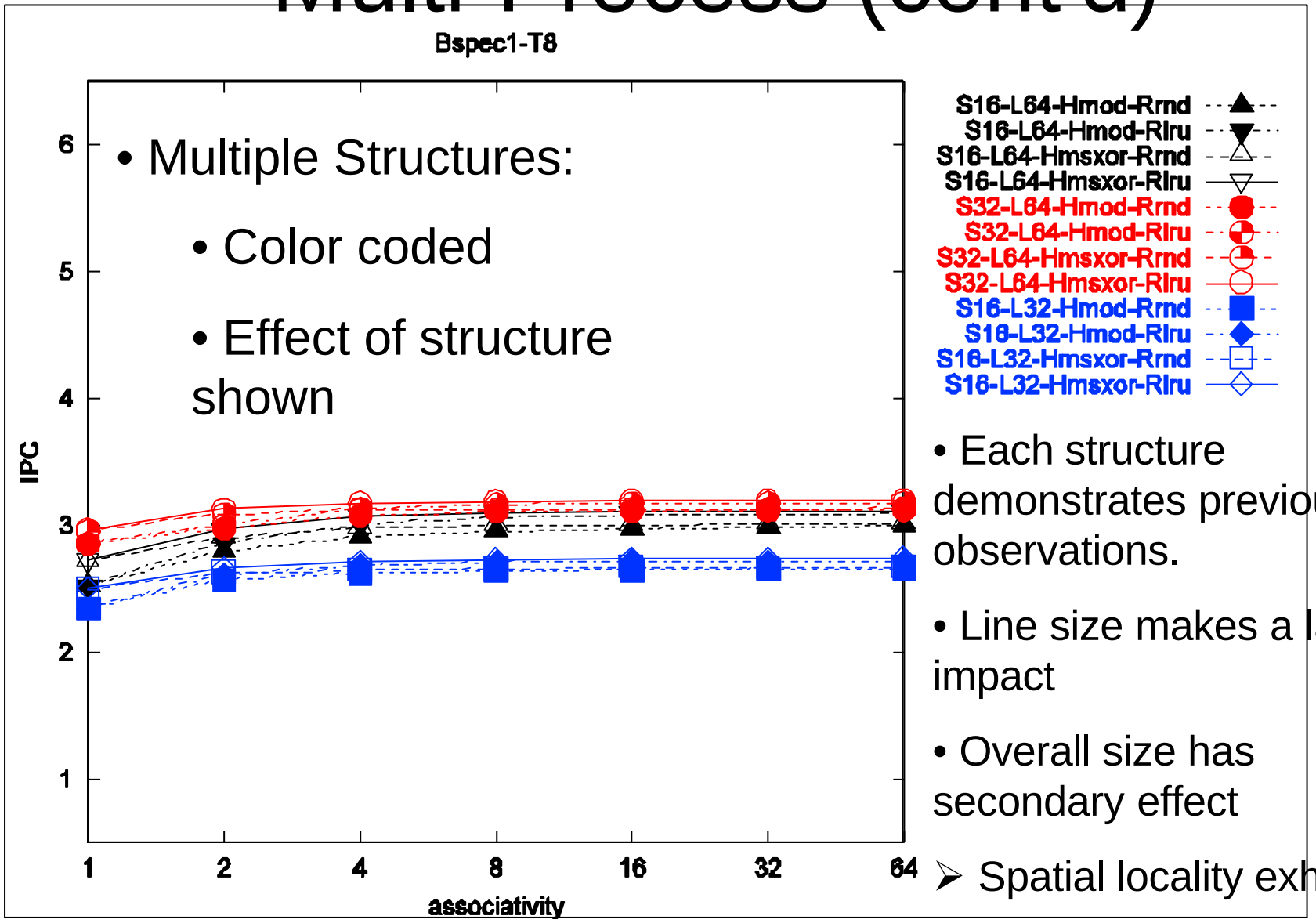

- Multi-process load
- Structure: 16Kb, 64B/line
- triangles: color=hash direction=rep.

- Hash func. important for low assoc.
- Rep. policy important for high assoc.
- Fast cross-over (4-way)
- LRU better than random
- Monotonic improvement with assoc.

➢ Classic temporal locality exhibited

- Saturates fast: Small overall change
  - Xor helps saturates faster

# Multi-Process (cont'd)

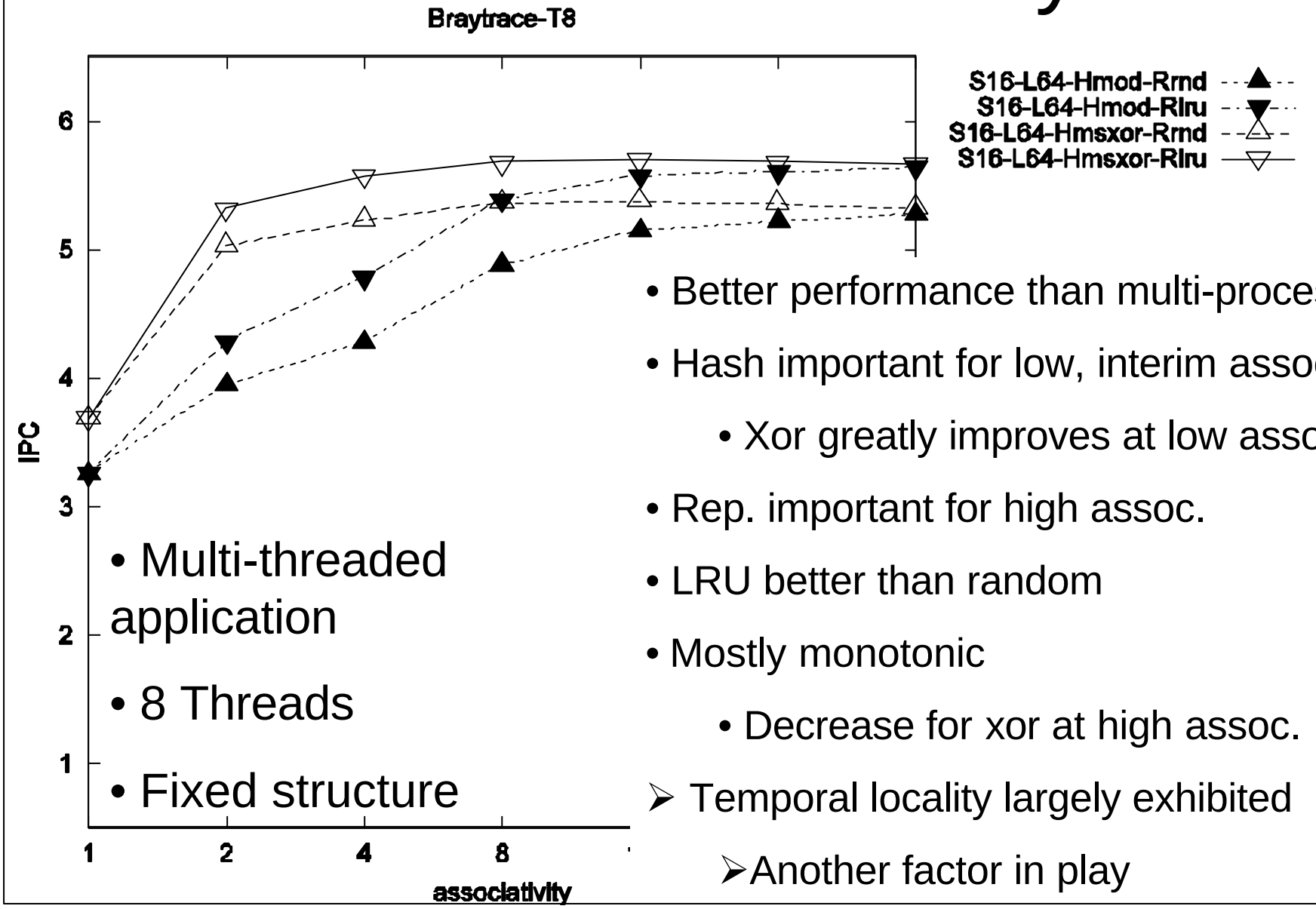



- Multiple Structures:
  - Color coded
  - Effect of structure shown

- Each structure demonstrates previous observations.
- Line size makes a large impact
- Overall size has secondary effect

➢ Spatial locality exhibited

# Multi-Threaded Load: Raytrace



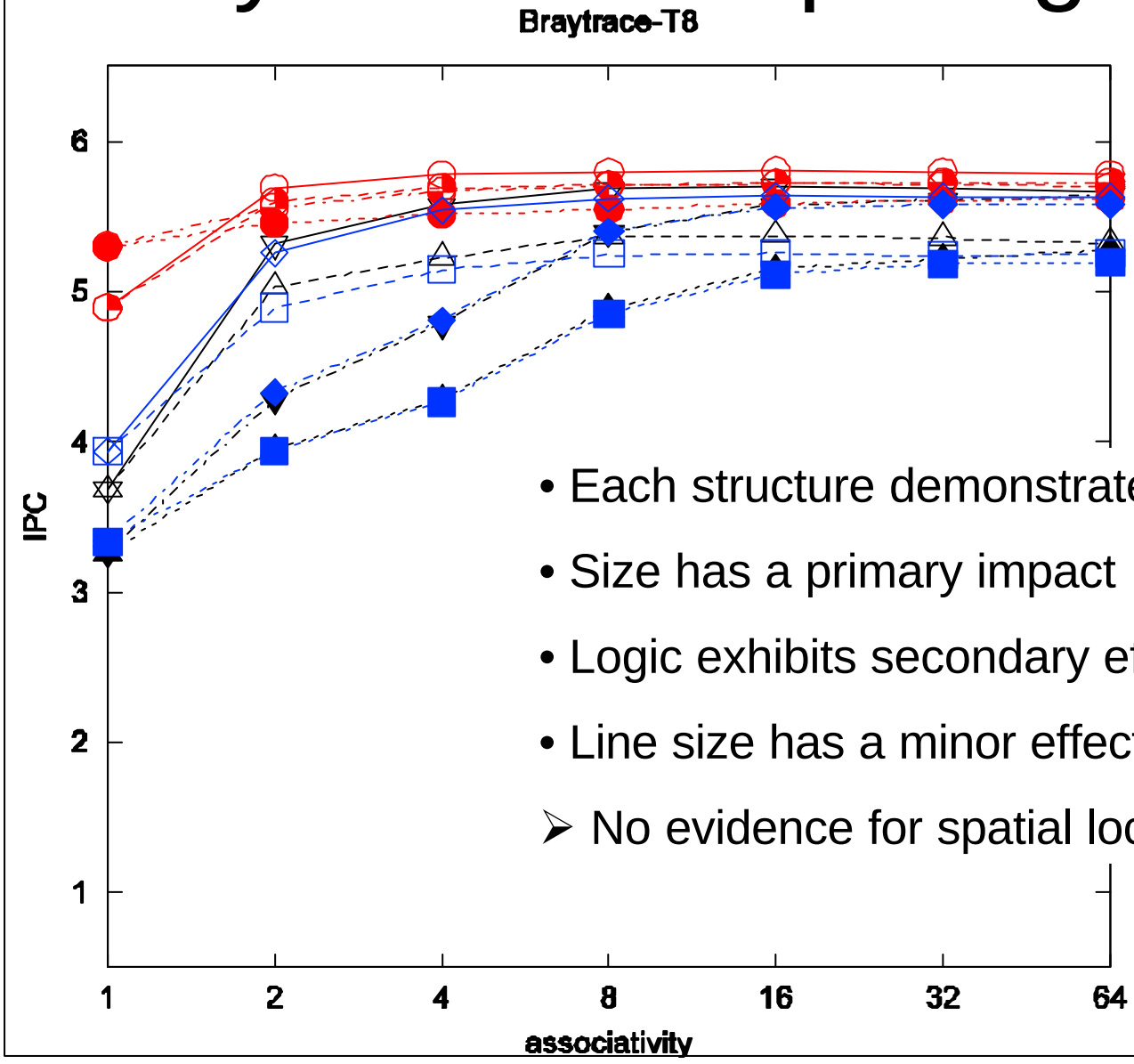

- Multi-threaded application
- 8 Threads
- Fixed structure

- Better performance than multi-process
- Hash important for low, interim assoc.
  - Xor greatly improves at low assoc.
- Rep. important for high assoc.
- LRU better than random
- Mostly monotonic
  - Decrease for xor at high assoc.

➢ Temporal locality largely exhibited

➢ Another factor in play

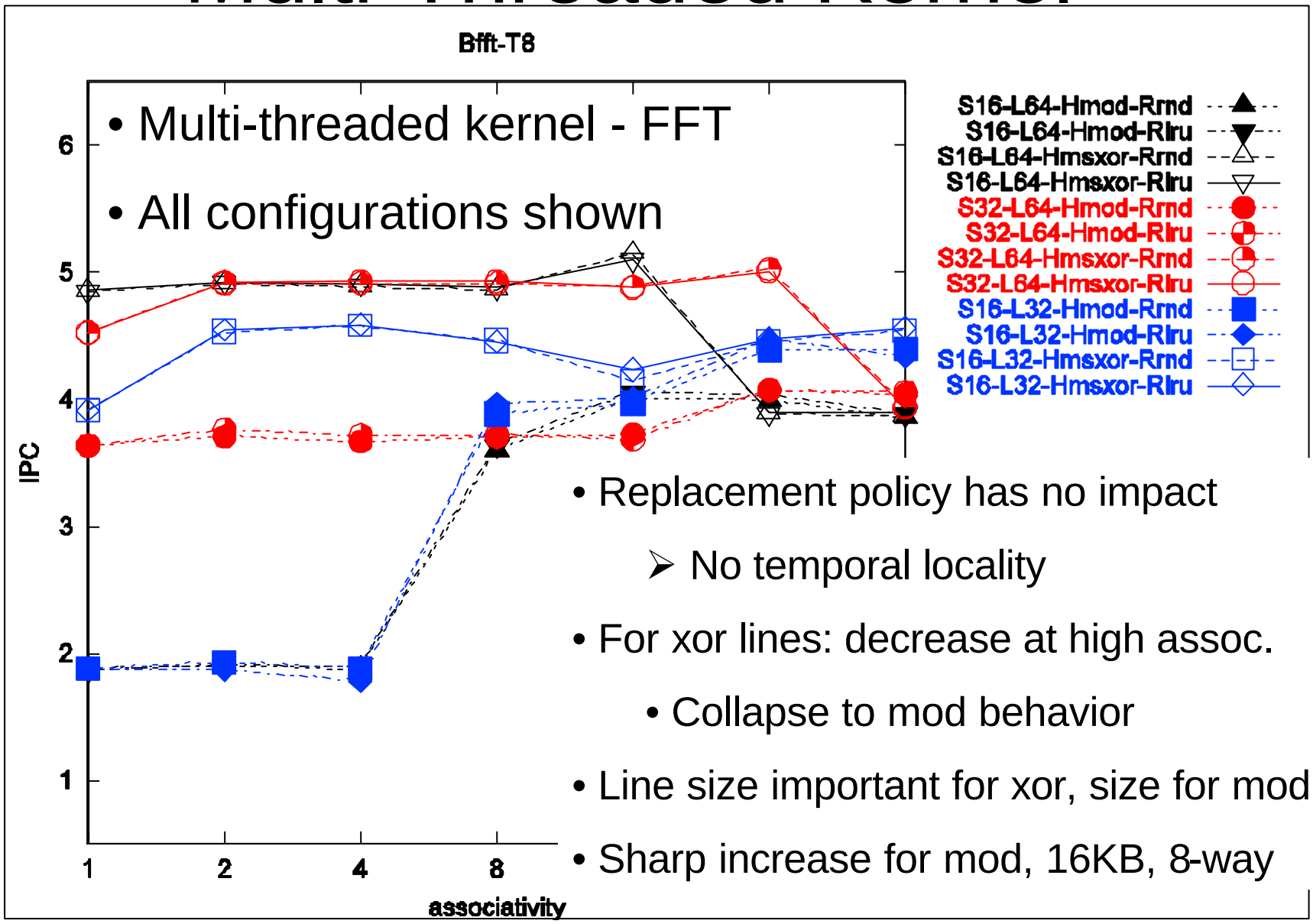
# Raytrace: Comparing Structure




- Each structure demonstrates previous observations.
- Size has a primary impact
- Logic exhibits secondary effect
- Line size has a minor effect

➢ No evidence for spatial locality

# Multi-Threaded Kernel




- Multi-threaded kernel - FFT
- All configurations shown

- Replacement policy has no impact
  - ➢ No temporal locality
- For xor lines: decrease at high assoc.
  - Collapse to mod behavior
- Line size important for xor, size for mod
- Sharp increase for mod, 16KB, 8-way

# Conclusions

- Multi-process loads exhibit classic access locality
- Multi-threaded loads exhibit “chaotic” factor
  - Small impact for raytrace
  - Only factor for kernels
  - Best dealt with using xor-hash for uniform spread of accesses between sets
- Higher associativity levels are not necessarily better
  - Xor hash enables fast performance increase
  - A 4 way set associative cache yields good results, negligible increase (if at all) beyond 8 way

# Acknowledgements


- Avi Mendelson, Intel and the Technion
- Dean M. Tullsen, UCSD
- Distributed Systems Laboratory, Technion

# The End

Questions?