Active Management of Cache Resources

Subramanian Ramaswamy

Computer Architecture and Systems Lab
School of Electrical and Computer Engineering
Georgia Institute of Technology

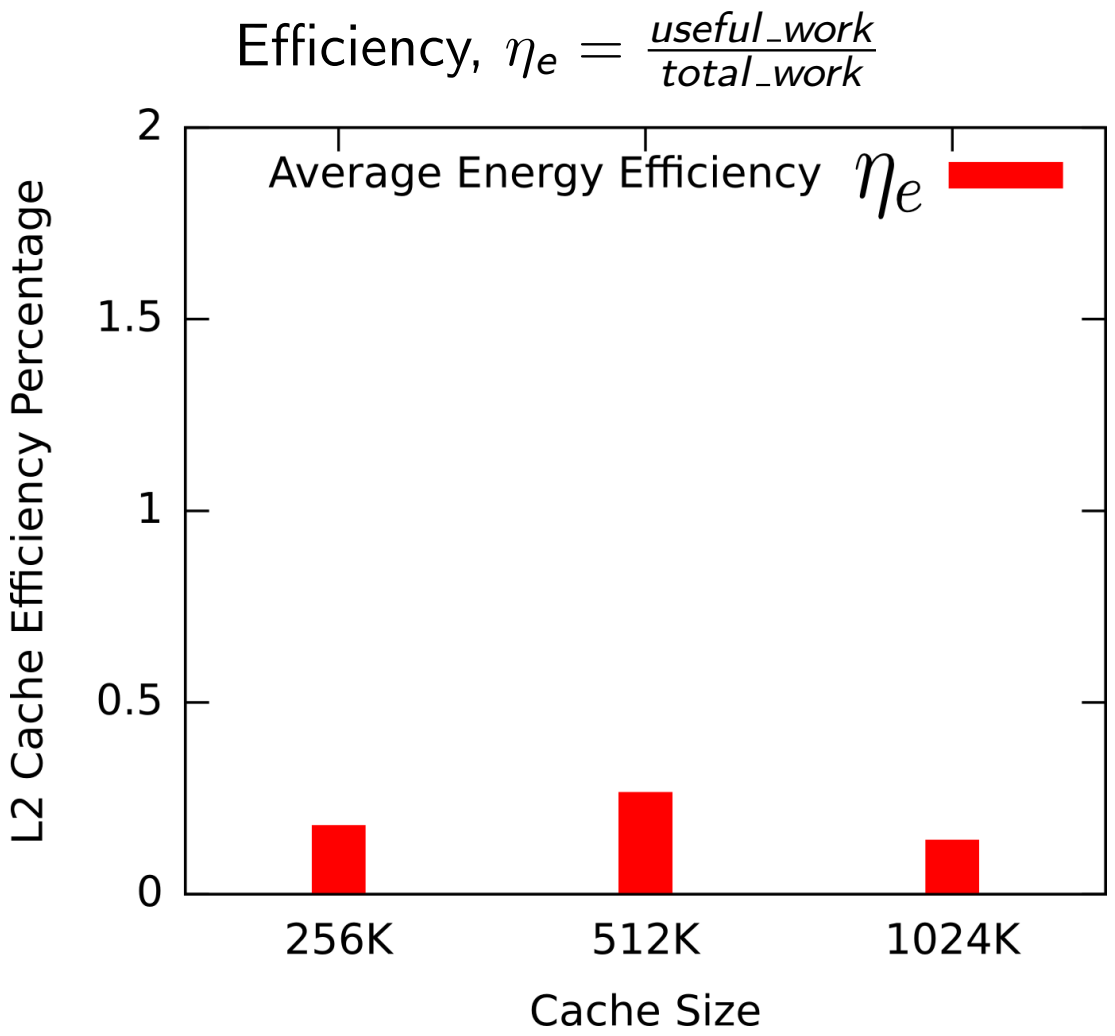
PhD Proposal

Advisor: Prof. Sudhakar Yalamanchili

September 14, 2007

Outline

Problem Focus

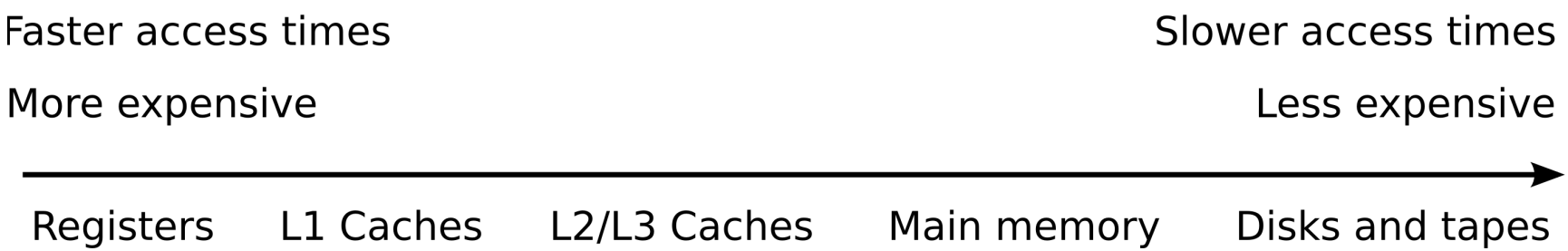


This research focuses on increasing data cache efficiency

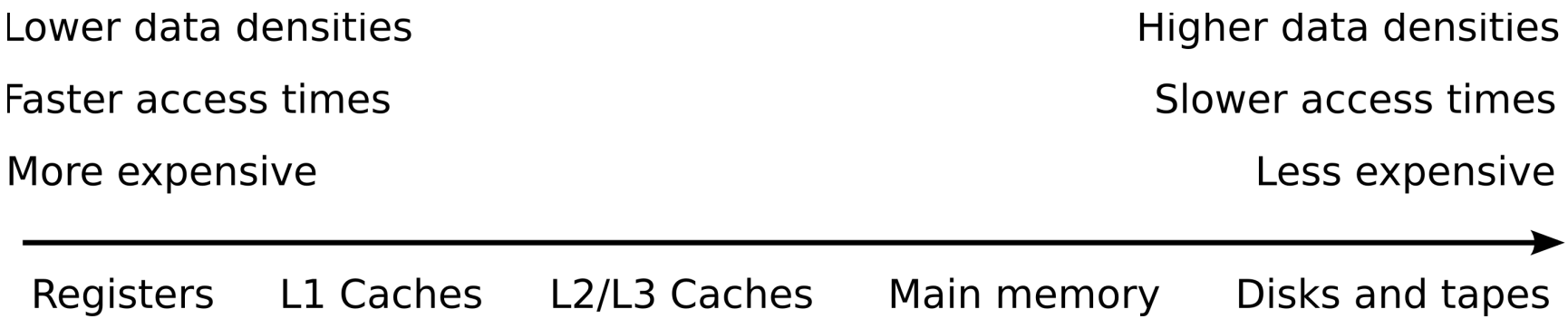
Caches 101


More expensive
Less expensive
Registers
L1 Caches
L2/L3 Caches
Main memory
Disks and tapes

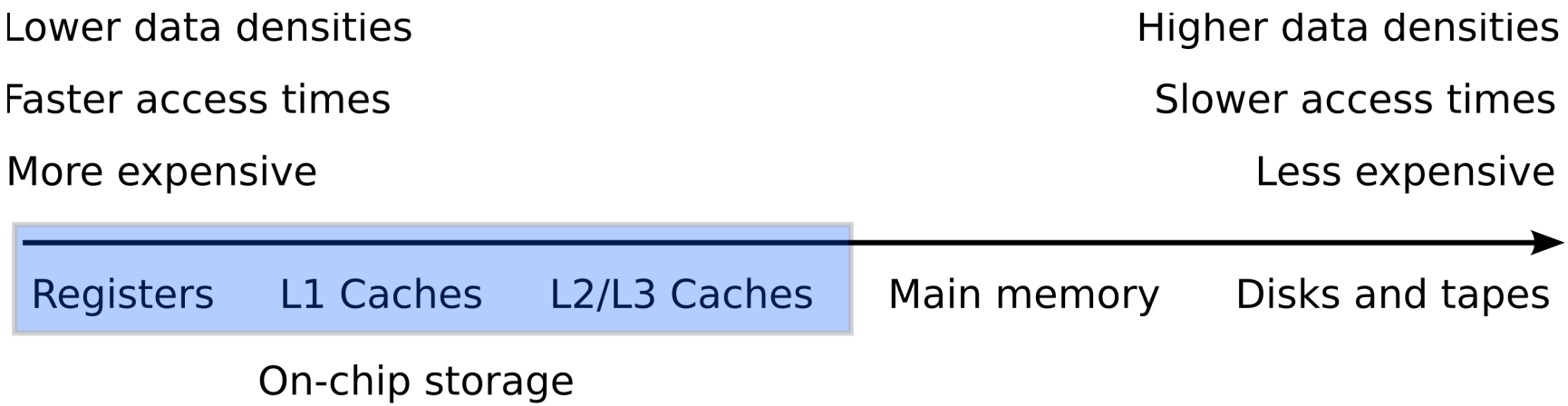
Caches 101


Faster access times
More expensive
Slower access times
Less expensive
Registers
L1 Caches
L2/L3 Caches
Main memory
Disks and tapes

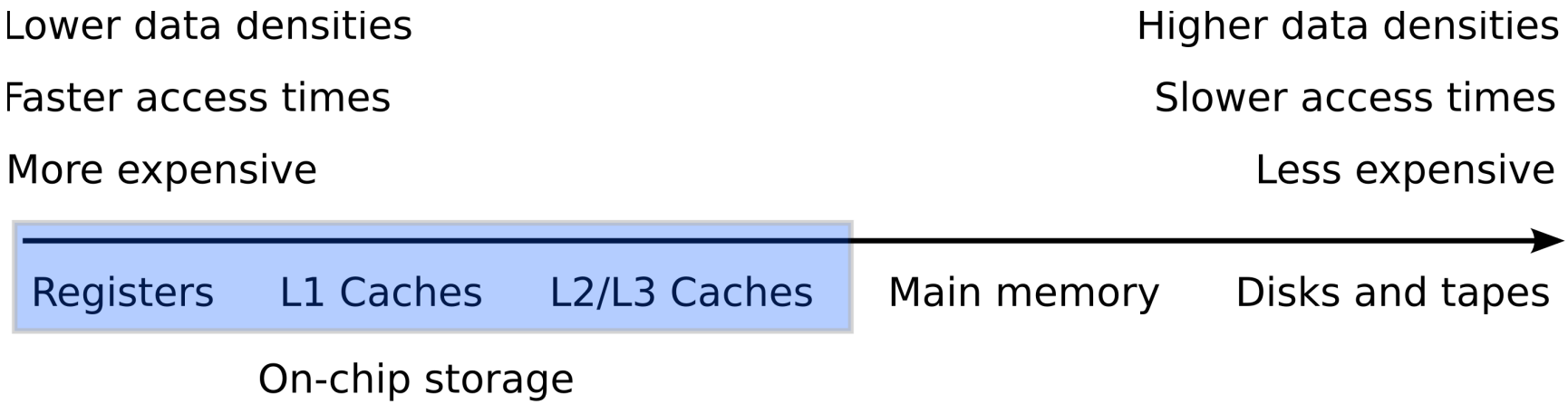
Caches 101


Lower data densities
Faster access times
More expensive
Higher data densities
Slower access times
Less expensive
Registers
L1 Caches
L2/L3 Caches
Main memory
Disks and tapes

Caches 101

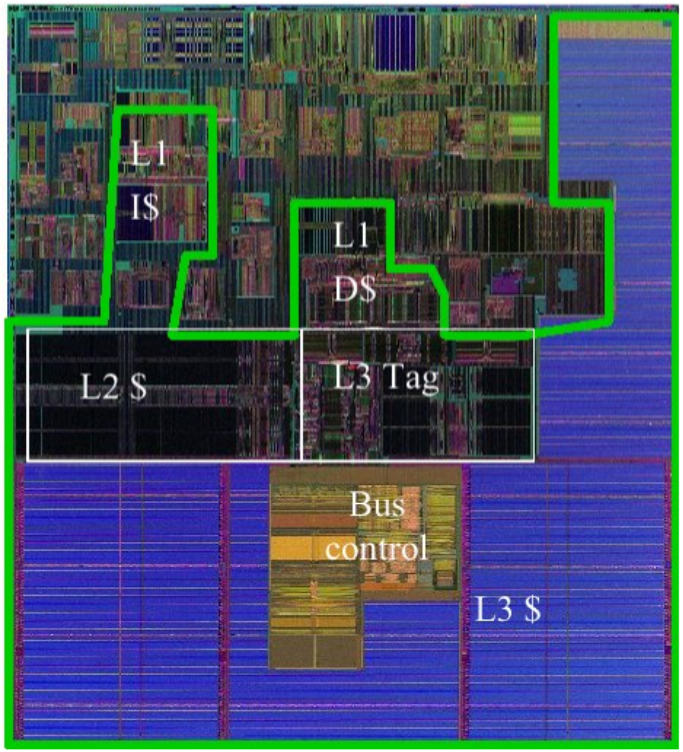

Lower data densities
Faster access times
More expensive
Higher data densities
Slower access times
Less expensive
Registers
L1 Caches
L2/L3 Caches
Main memory
Disks and tapes
On-chip storage

Caches 101



Caches exploit **spatial** and **temporal** locality to minimize off-chip accesses

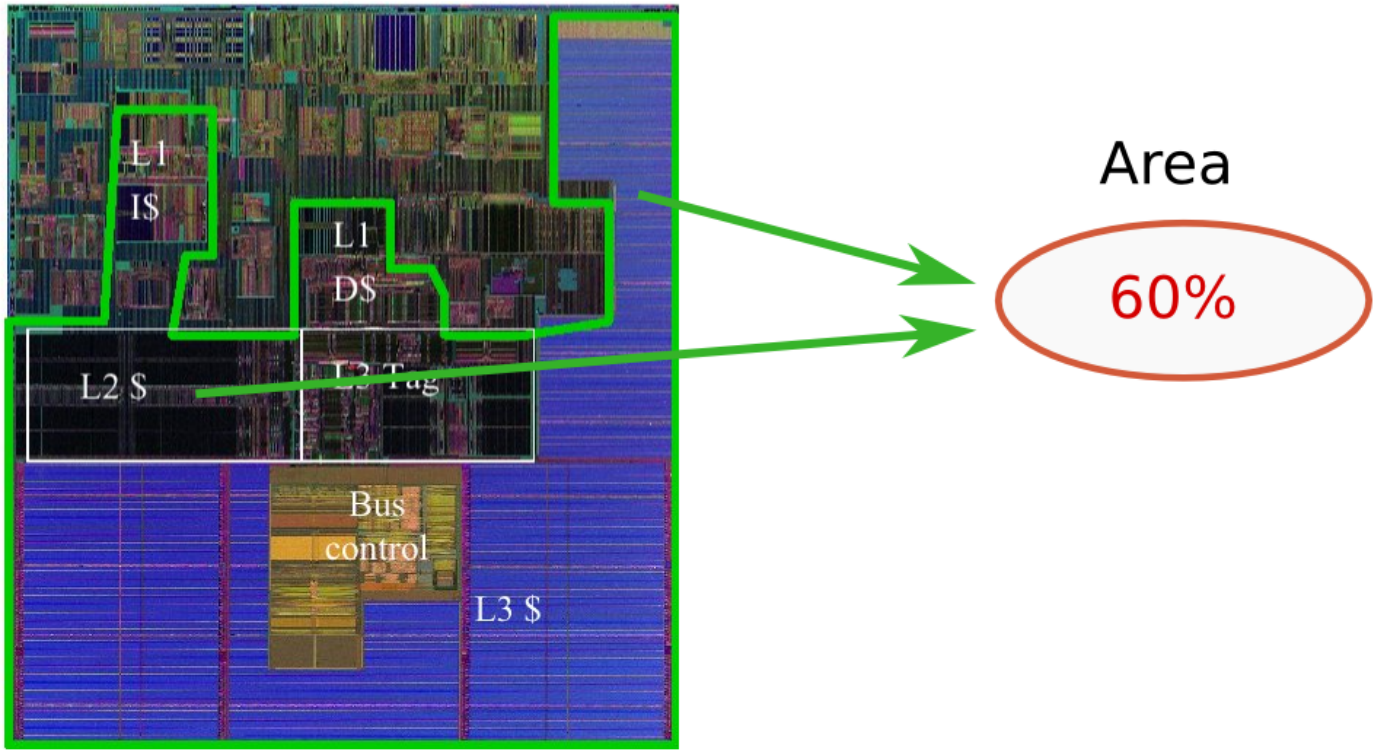
Resource Consumption of Caches



Intel Itanium 2 (Source: Intel)

- Processor-memory gap → increasing cache sizes

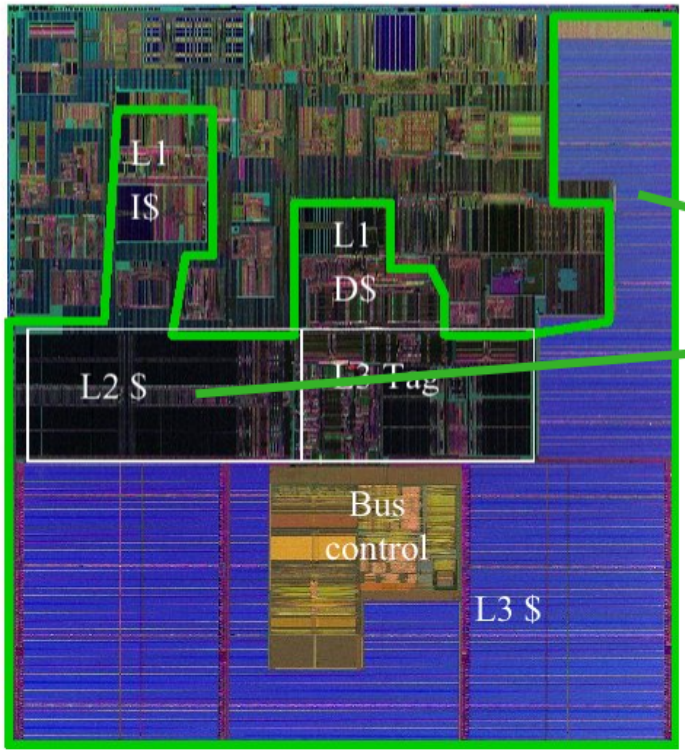
Resource Consumption of Caches



Intel Itanium 2 (Source: Intel)

- Processor-memory gap → increasing cache sizes

Resource Consumption of Caches



Intel Itanium 2 (Source: Intel)

Area

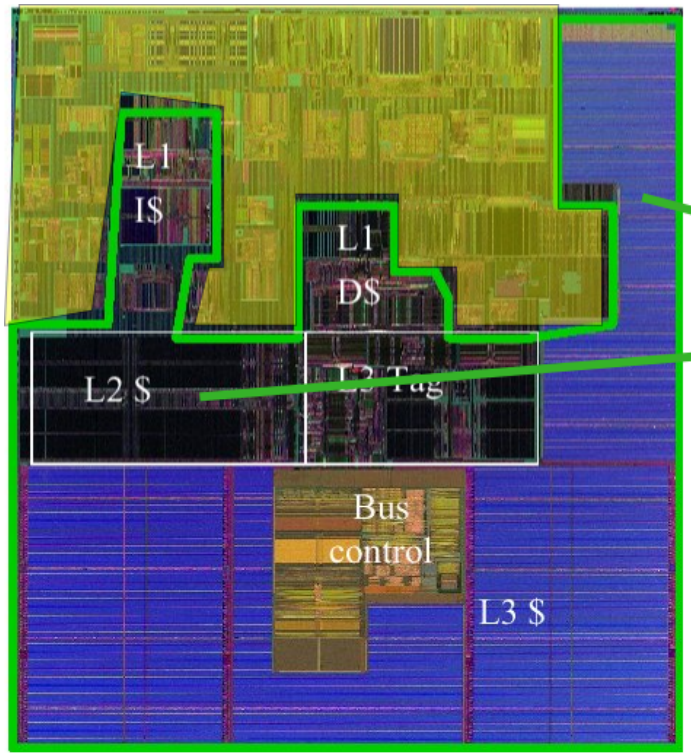
60%

Energy

15-30%

- Processor-memory gap → increasing cache sizes

Resource Consumption of Caches



Intel Itanium 2 (Source: Intel)

Area

60%

Energy

15-30%

- Blocks doing actual computation devoted less resources

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Technology Challenges

- More caches means less for processing elements
  - Frequency ramping has ended → shift from increasing clock frequency to number of cores
- Leakage current increases exponentially with successive technology generations
  - Caches are major contributors to leakage energy
- Within-die variations at DSM technology nodes → die yield challenges
  - Cache SRAM cells are susceptible to multiple failure modes

Why Care About Efficiency?

- Smaller cache footprint for $\approx$ same cache performance

Why Care About Efficiency?

- Smaller cache footprint for $\approx$ same cache performance
- Power and energy advantages

Why Care About Efficiency?

- Smaller cache footprint for $\approx$ same cache performance
- Power and energy advantages
- Release die area for alternative enhancement vehicles

Why Care About Efficiency?

- Smaller cache footprint for $\approx$ same cache performance
- Power and energy advantages
- Release die area for alternative enhancement vehicles
- Higher performance with same or smaller sized caches

Why Care About Efficiency?

- Smaller cache footprint for $\approx$ same cache performance
- Power and energy advantages
- Release die area for alternative enhancement vehicles
- Higher performance with same or smaller sized caches
- Higher yield as performance impact of faults can be absorbed

Quantifying Resource Usage

Concepts

- A cache line is **live** at a clock cycle if it contains data that will be used prior to eviction; **dead** otherwise
- A cache line is **active** if it is "ON"
- A cache line contributes a **live**, **dead**, or **inactive** cycle at every clock cycle

Quantifying Resource Usage

Concepts

- A cache line is **live** at a clock cycle if it contains data that will be used prior to eviction; **dead** otherwise
- A cache line is **active** if it is "ON"
- A cache line contributes a **live**, **dead**, or **inactive** cycle at every clock cycle

Utilization

$$\eta_u = \frac{\sum_{i=0}^{i=L-1} live\_cycles_{line_i}}{\sum_{i=0}^{i=L-1} active\_cycles_{line_i}}$$

Measures the percentage of the cache that is **"live"** averaged over time

Cache Efficiency

Cache Efficiency

Performance Efficiency

$$\eta_p = \frac{\eta_u * t_c}{t_c + m * t_p}$$

t_c:cache access time, m: miss rate, t_p:miss penalty

Cache Efficiency

Performance Efficiency

$$\eta_p = \frac{\eta_u * t_c}{t_c + m * t_p}$$

t_c:cache access time, m: miss rate, t_p:miss penalty

Energy Efficiency

$$\eta_e = \frac{sw\_en_{hits}}{sw\_en_{hits} + sw\_en_{misses} + leak\_en}$$

Cache Efficiency

Performance Efficiency

$$\eta_p = \frac{\eta_u * t_c}{t_c + m * t_p}$$

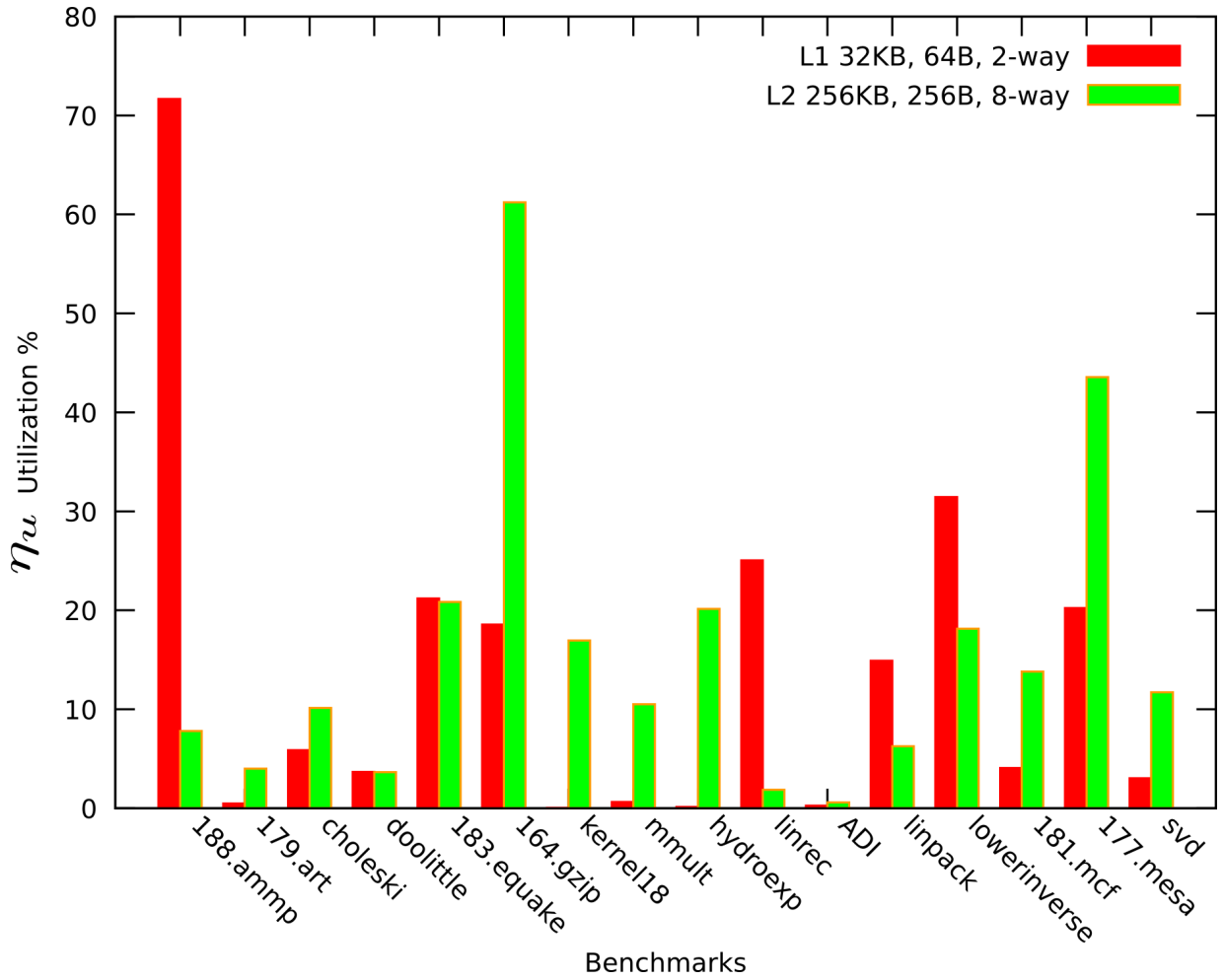
t_c:cache access time, m: miss rate, t_p:miss penalty

Energy Efficiency

$$\eta_e = \frac{sw\_en_{hits}}{sw\_en_{hits} + sw\_en_{misses} + leak\_en}$$

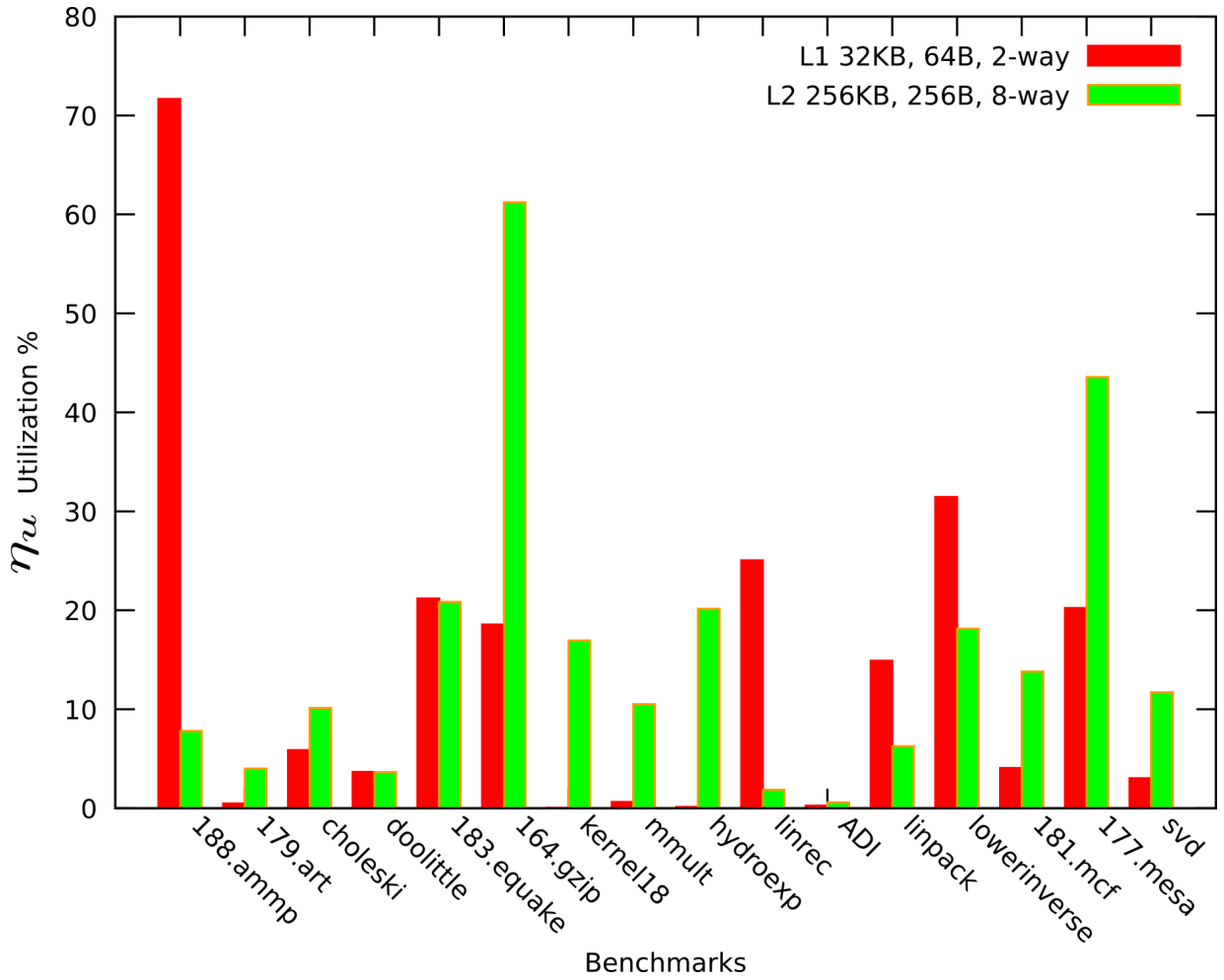
- Utilization captures the temporal residency of live data in the cache
- Performance efficiency captures how well the residency of live data in the cache is exploited
- Energy efficiency is the percentage of **useful** work to total work (total energy)

Utilization Results



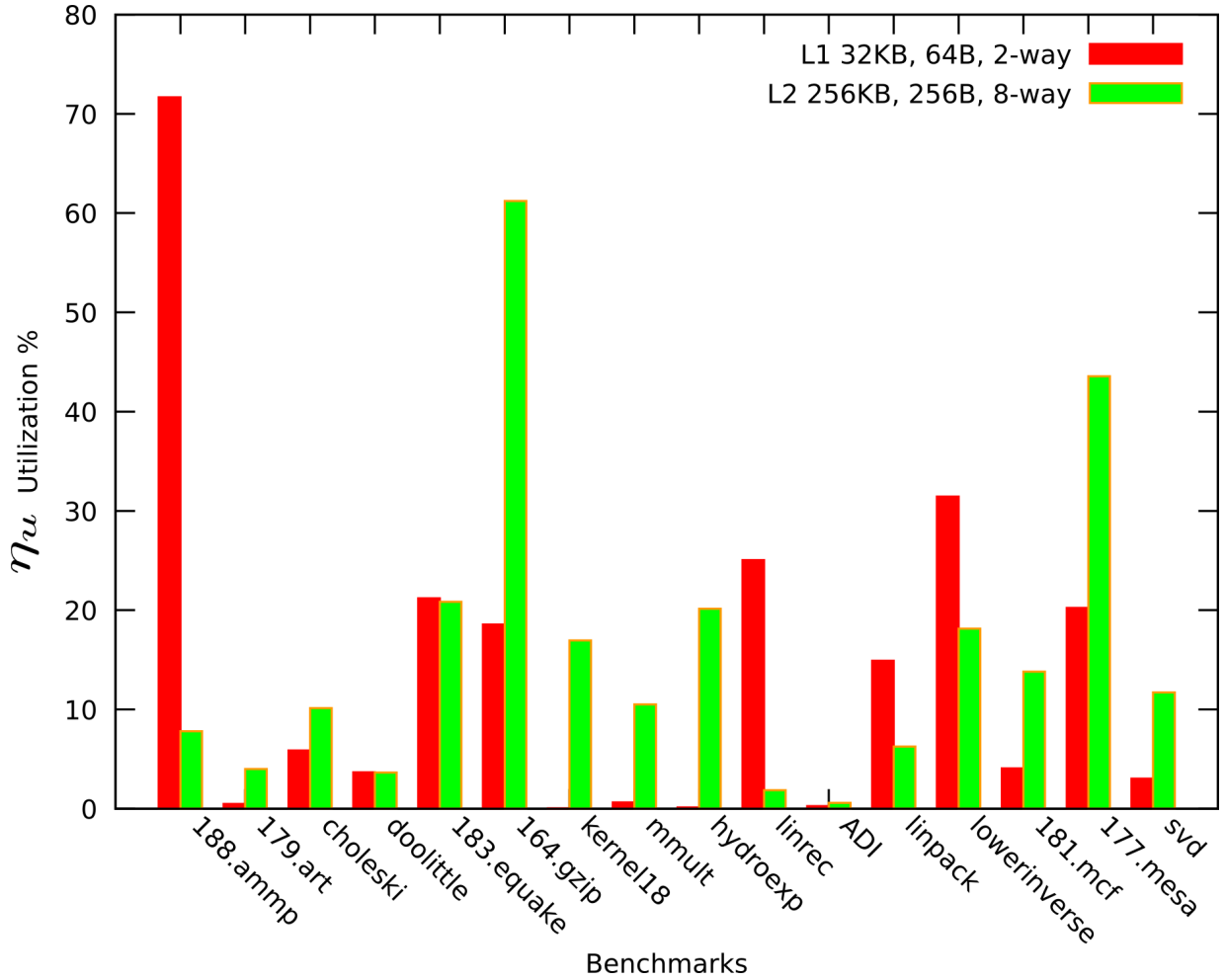
- η_u averages $< 20\% \rightarrow$ dead cycles dominate

Utilization Results



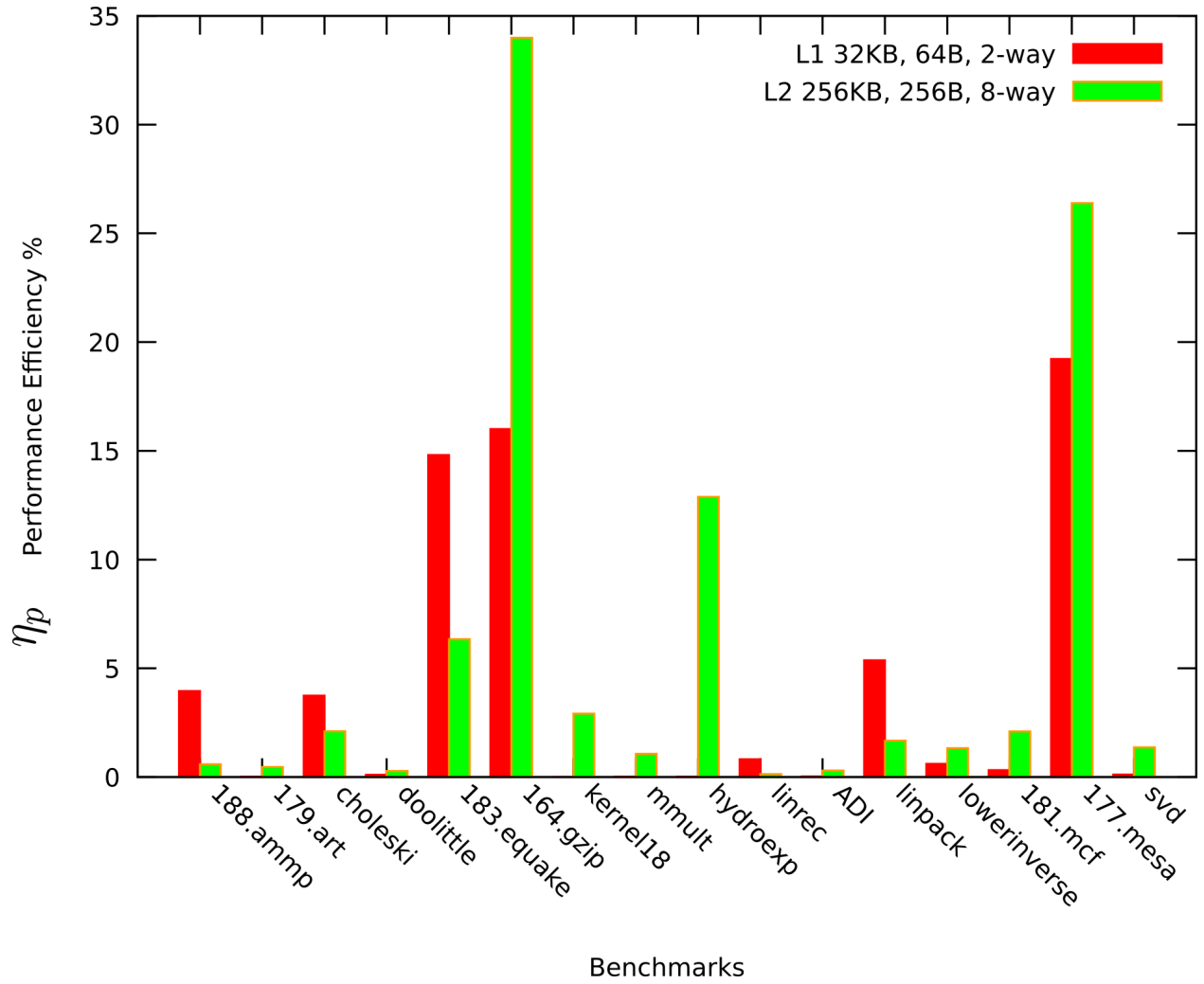
- Associativity and size increases have low impact

Utilization Results



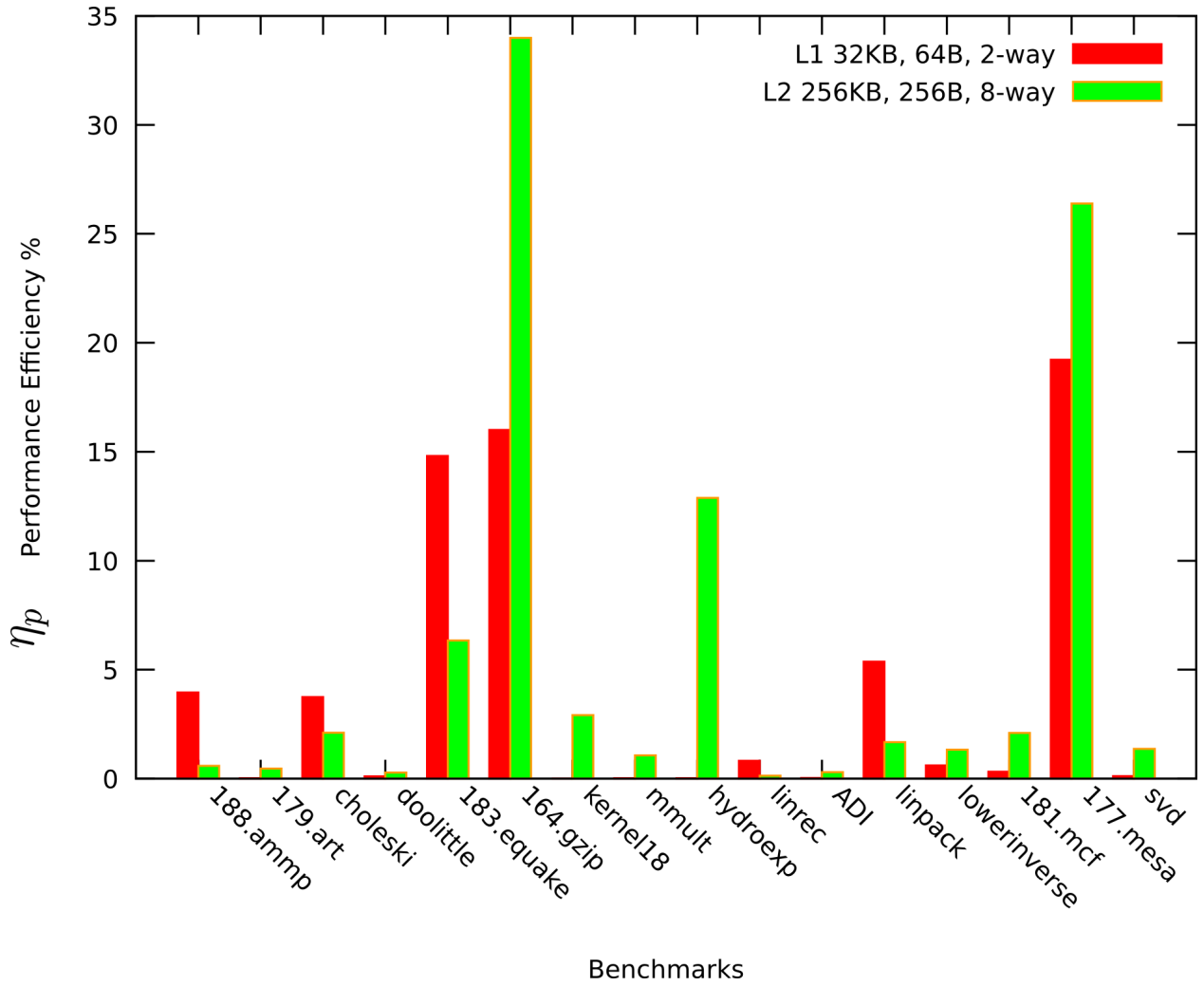
- *Word utilization* is lower than line utilization

Performance Efficiency



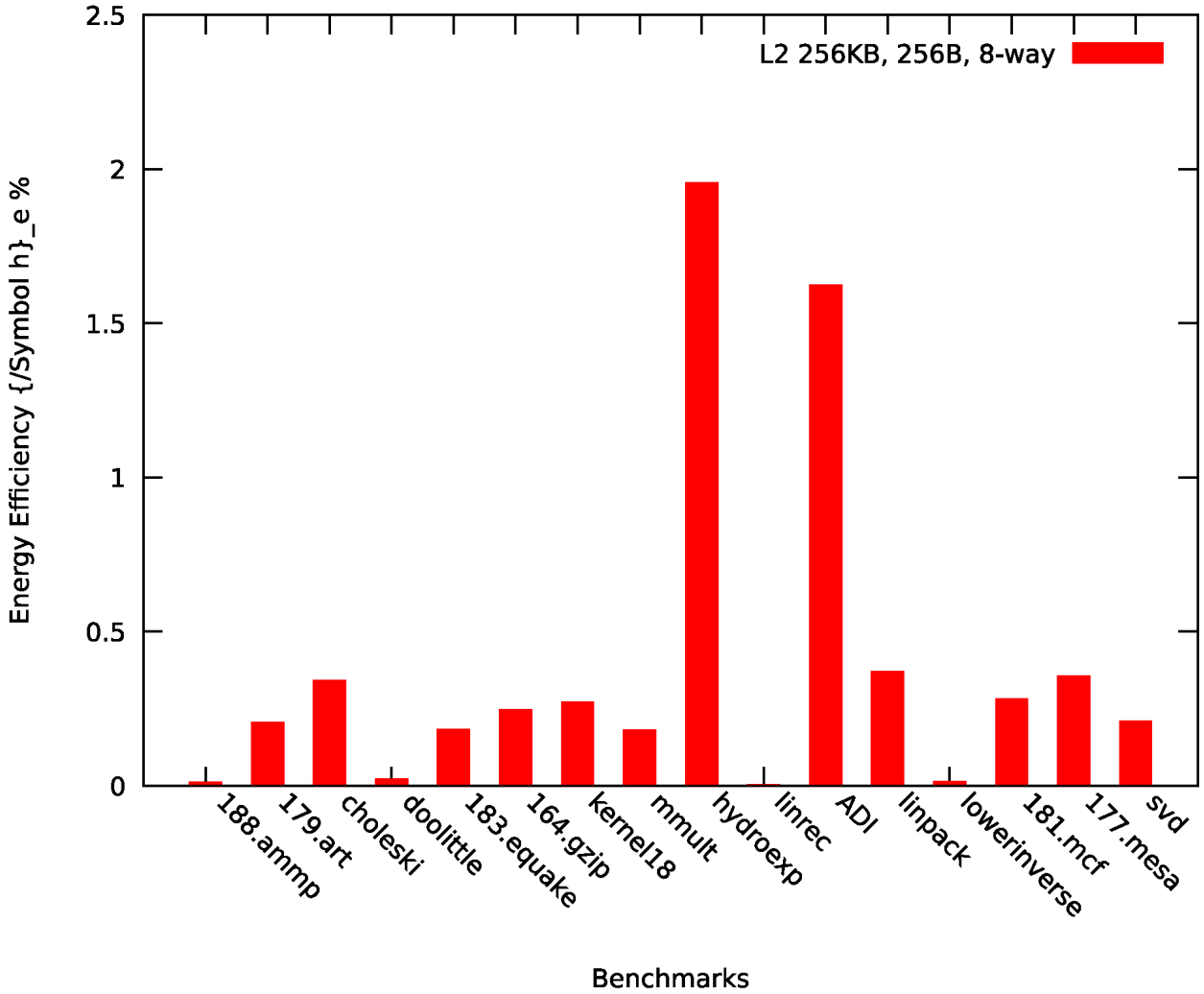
- η_u is an upper bound for η_p

Performance Efficiency



- η_p average is $< 15\%$

Energy Efficiency



- η_e average is $< 0.5\%$

Sources of Inefficiency

Sources of Inefficiency

- Predominance of leakage energy

Sources of Inefficiency

- Predominance of leakage energy
- Every miss contributes 200 cycles $\rightarrow$ 1024 cache lines contributing to leakage for an additional 200 cycles

Sources of Inefficiency

- Predominance of leakage energy
- Every miss contributes 200 cycles $\rightarrow$ 1024 cache lines contributing to leakage for an additional 200 cycles
- PVT variations $\rightarrow$ faults lead to performance impact

Sources of Inefficiency

- Predominance of leakage energy
- Every miss contributes 200 cycles $\rightarrow$ 1024 cache lines contributing to leakage for an additional 200 cycles
- PVT variations $\rightarrow$ faults lead to performance impact
- Time varying behavior of programs, e.g., changing program memory *footprint*

Sources of Inefficiency

- Predominance of leakage energy
- Every miss contributes 200 cycles $\rightarrow$ 1024 cache lines contributing to leakage for an additional 200 cycles
- PVT variations $\rightarrow$ faults lead to performance impact
- Time varying behavior of programs, e.g., changing program memory *footprint*
- Manner in which memory lines share the cache

Impediment to Increasing Efficiency: Passive Cache Management

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

- Access patterns have to exhibit spatial/temporal locality

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

- Access patterns have to exhibit spatial/temporal locality
- Program footprints not exploited

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

- Access patterns have to exhibit spatial/temporal locality
- Program footprints not exploited
- Current philosophy $\longrightarrow$ you need a big net to catch a few small fish

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

- Access patterns have to exhibit spatial/temporal locality
- Program footprints not exploited
- Current philosophy $\longrightarrow$ you need a big net to catch a few small fish

Impediment to Increasing Efficiency: Passive Cache Management

Cache designs are passive and program agnostic

- Access patterns have to exhibit spatial/temporal locality
- Program footprints not exploited
- Current philosophy $\longrightarrow$ you need a big net to catch a few small fish

- A design that has remained relatively unchanged over the last few decades

Thesis Statement

Active management of cache resources increases efficiency

Thesis Statement

Active management of cache resources increases efficiency

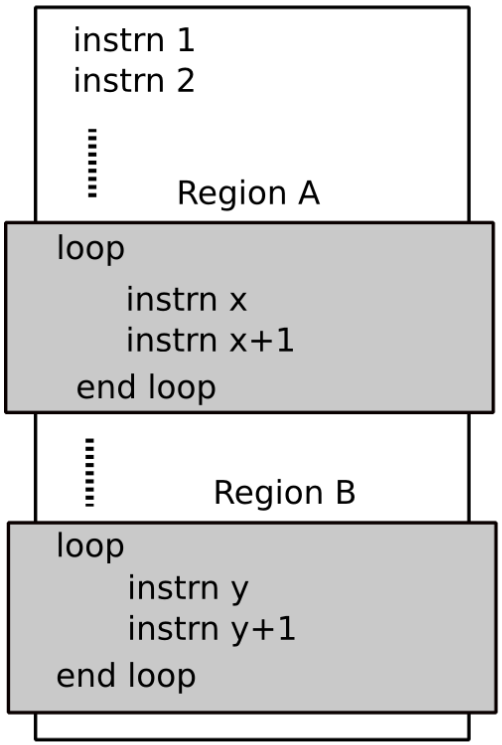
- Tune the cache by **sizing** and **shaping**

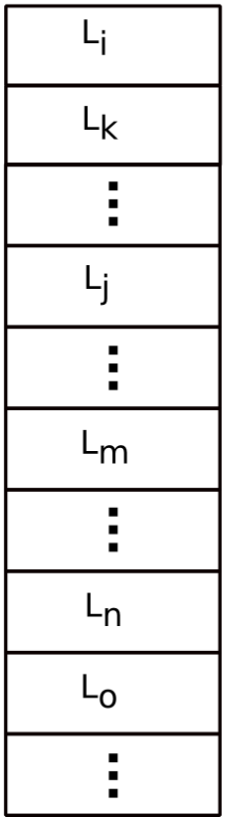
Thesis Statement

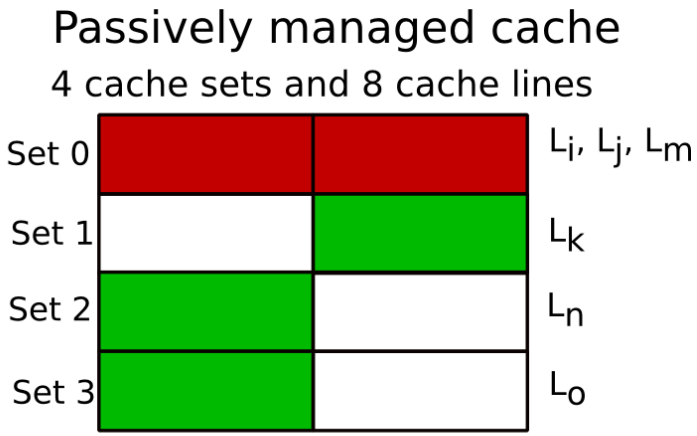
Active management of cache resources increases efficiency

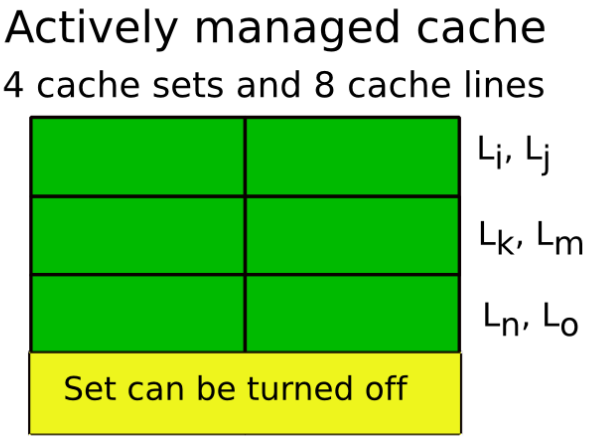
- Tune the cache by **sizing** and **shaping**
- Improve sharing of cache among main memory lines

Active Cache Management









Optimize the mapping of the program memory **footprint** in the cache

Managing Cache Resources for Improved Cache Sharing

8-line Main Memory

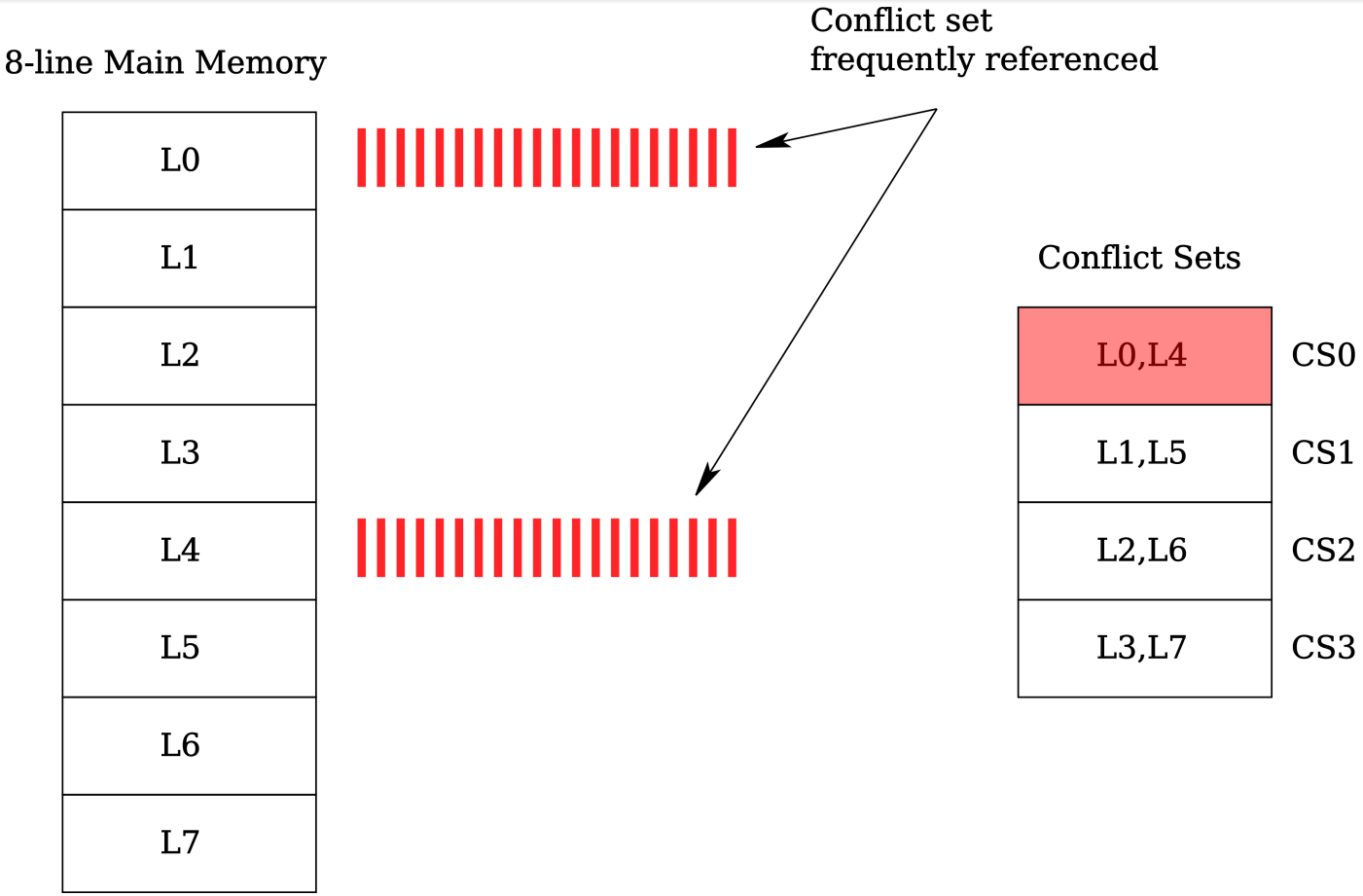
L0
L1
L2
L3
L4
L5
L6
L7

Conflict Sets

L0,L4	CS0
L1,L5	CS1
L2,L6	CS2
L3,L7	CS3

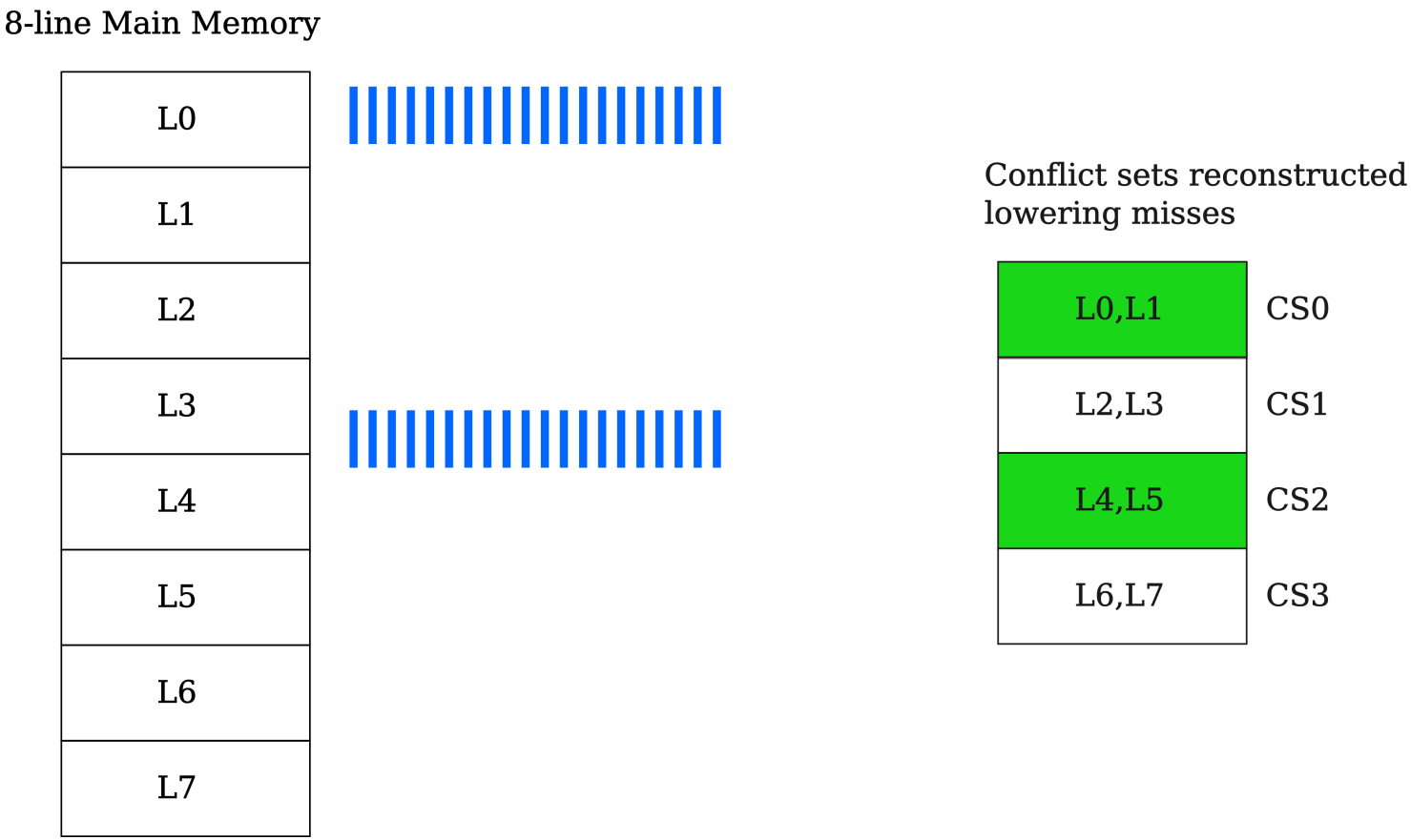
- **Modulo** placement maps memory line L_i to cache set $L_i mod S$
- Main memory lines are divided into **conflict sets**

Managing Cache Resources for Improved Cache Sharing



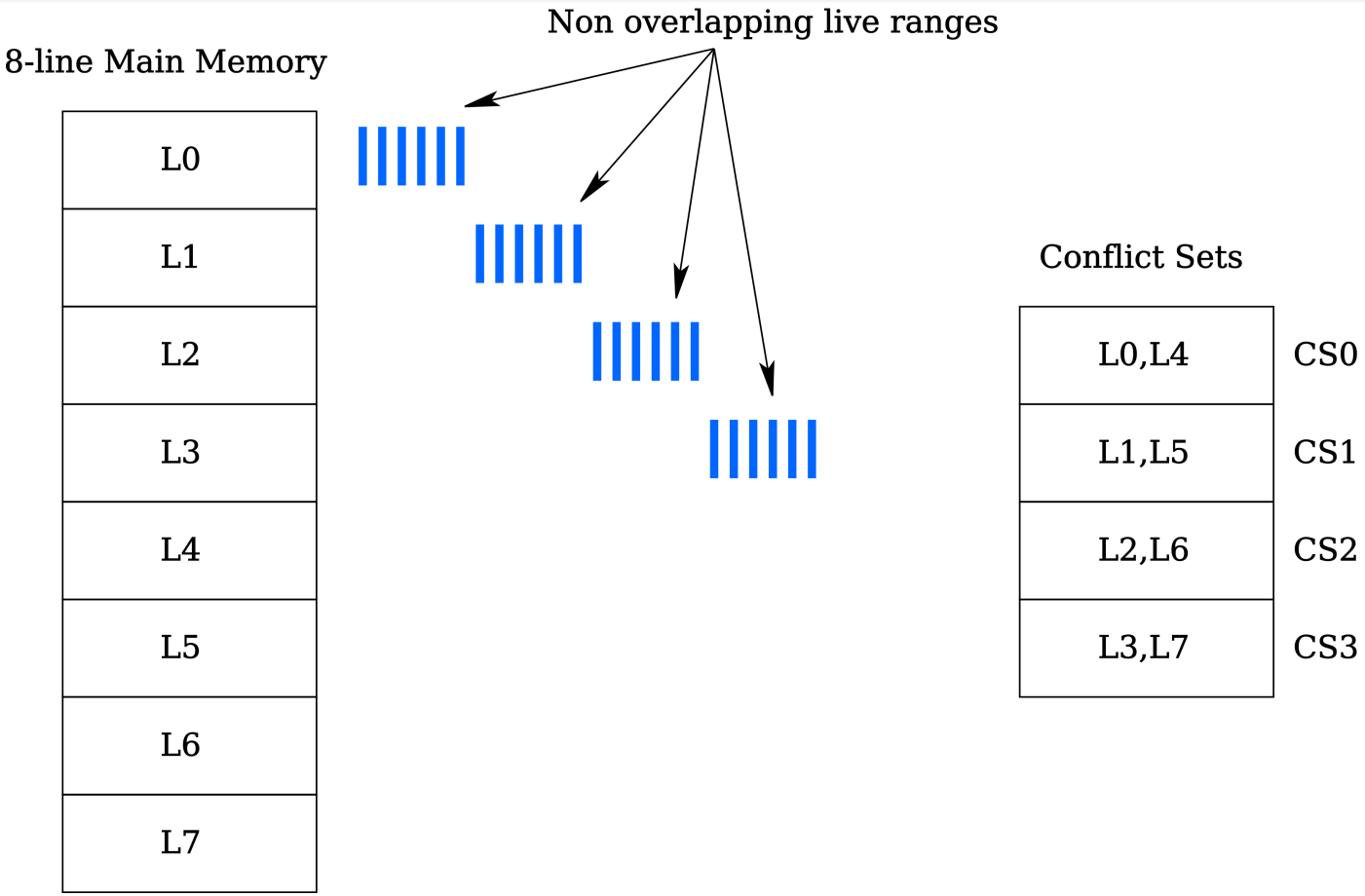
- References to *CS*0 thrash, while other conflict sets are not accessed

Managing Cache Resources for Improved Cache Sharing



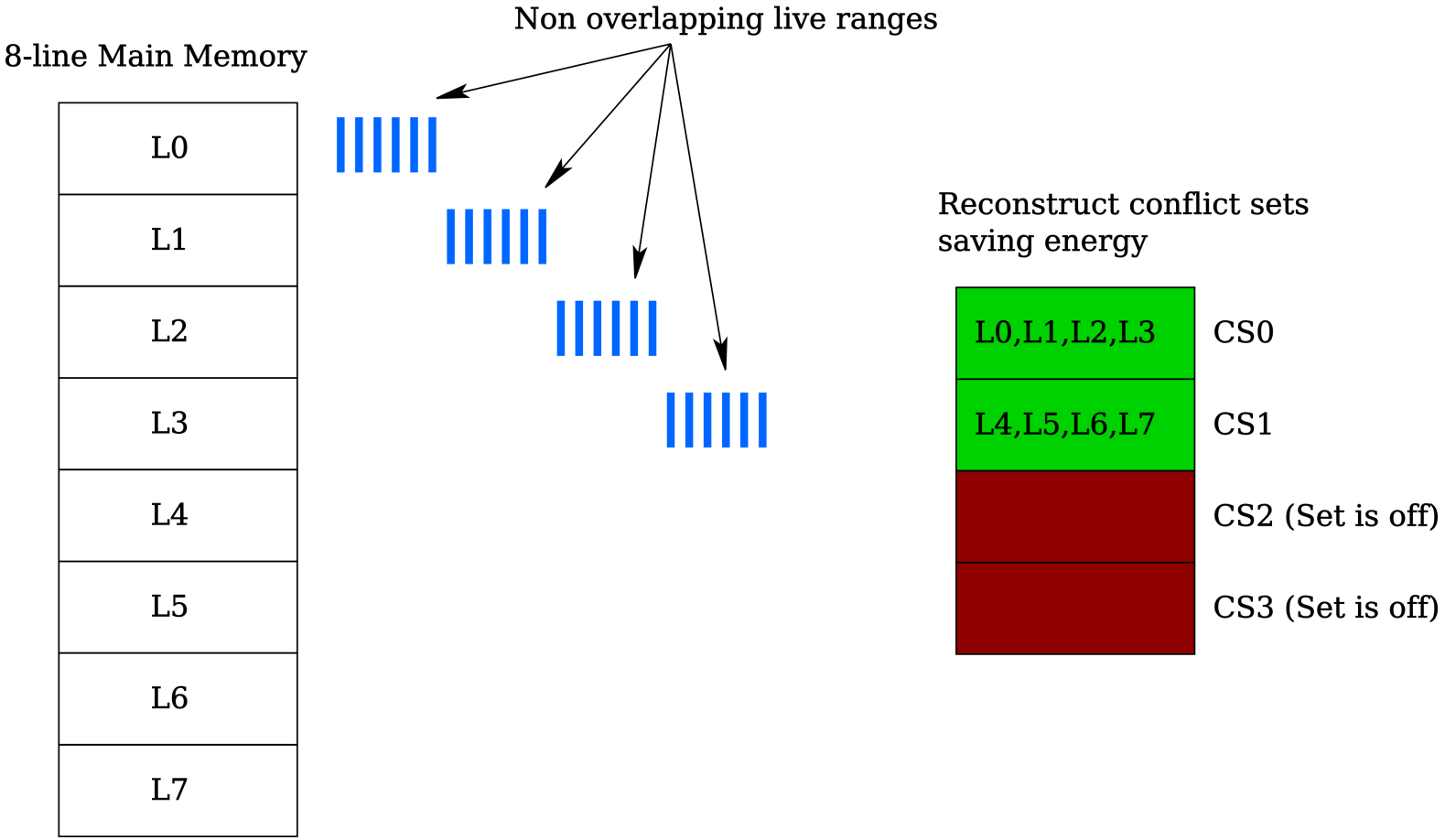
- Adapt the **placement** and reconstruct **conflict sets** for **better sharing** reducing contention

Cache Tuning



- Memory lines may exhibit non-overlapping live ranges

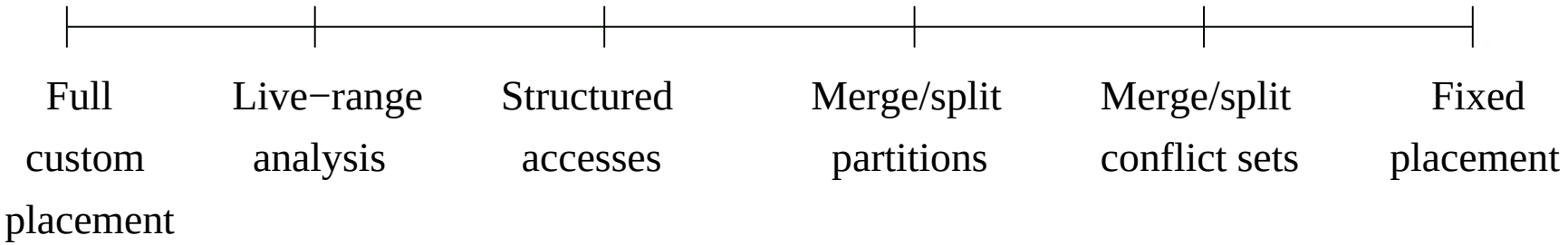
Cache Tuning



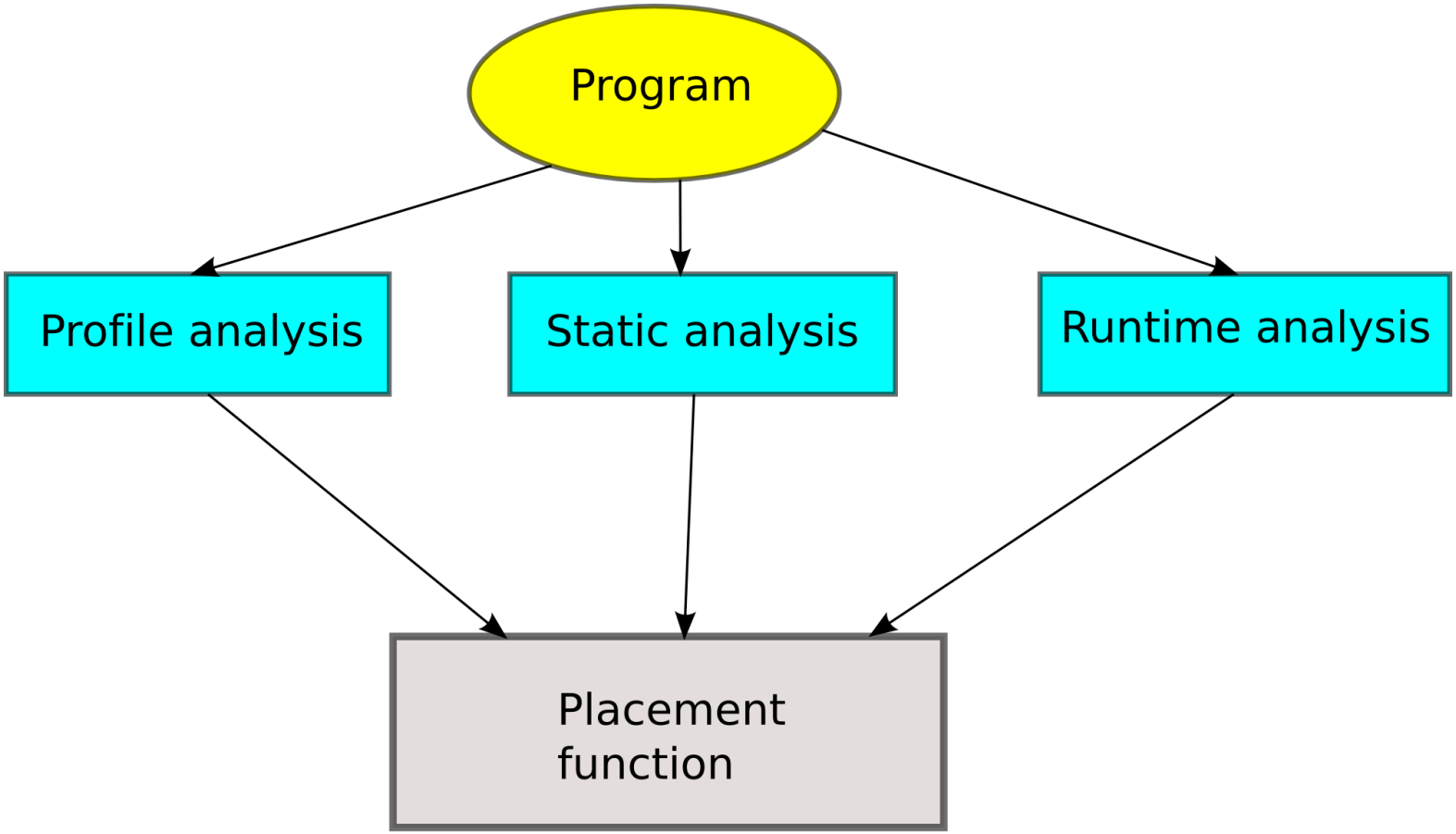
- Exploit **liveness** to **size** and **shape** caches
- Every memory line maps to an **active** cache set
- *Shaping* is a natural mechanism for improving fault-tolerance

The Optimization Problem

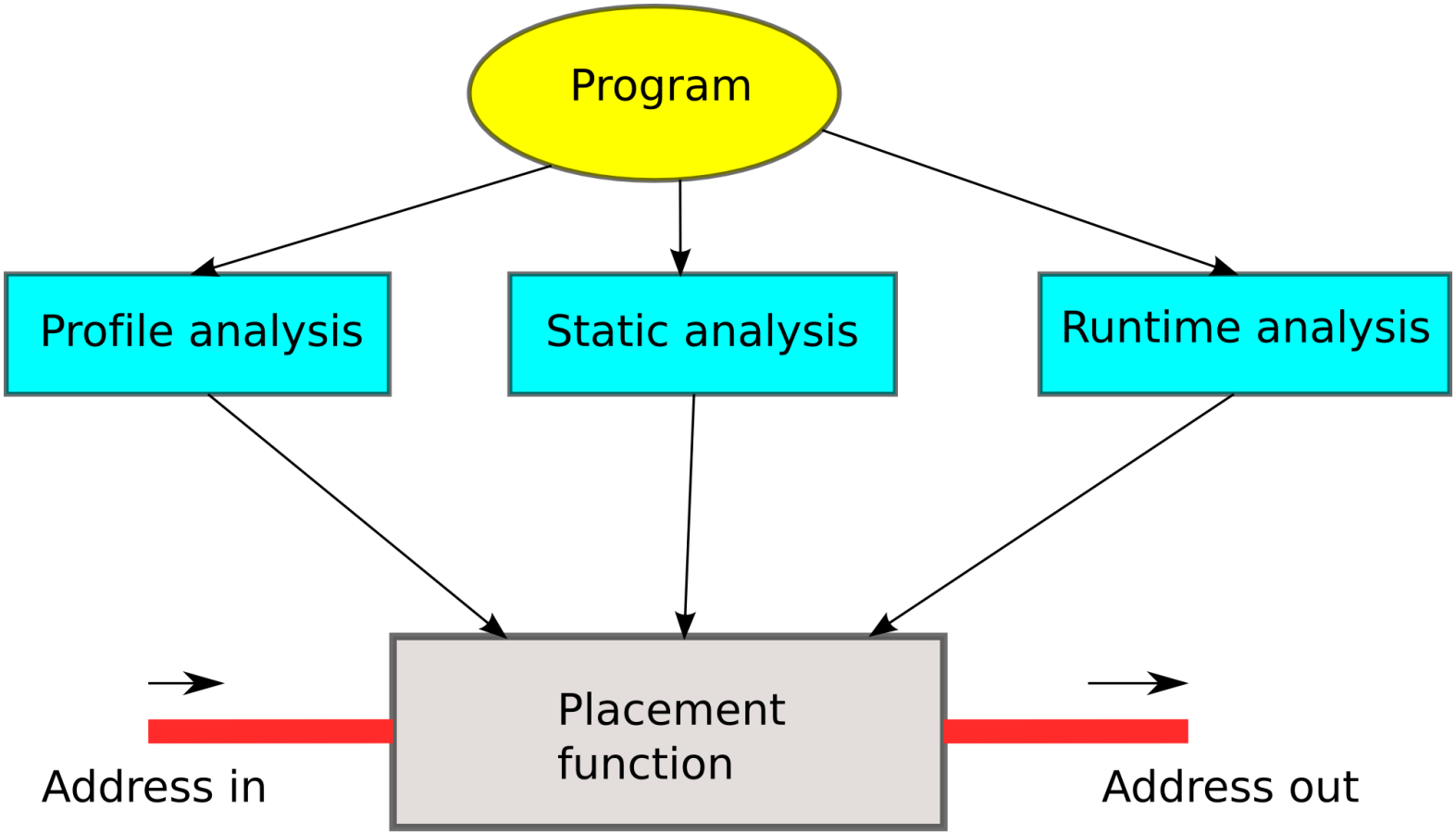
Given a memory access pattern, for a cache with S sets $(s_0, s_1, \ldots s_{S-1})$ and M main memory lines $(L_0, L_1 \ldots L_{M-1})$, compute a placement function, $placement(L_i) = S_j$ such that cache utilization/efficiency is maximized



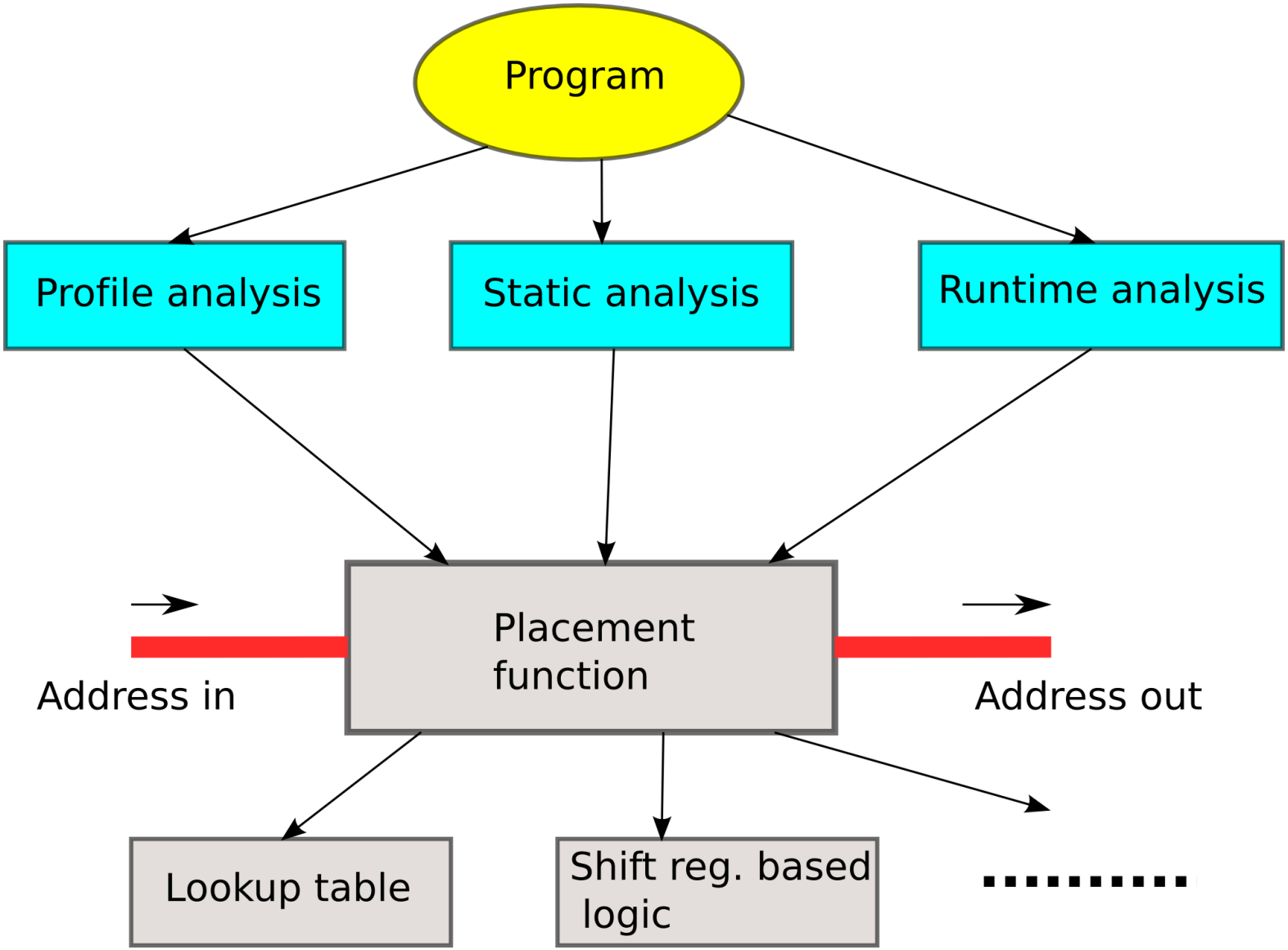
Scope of the Research



Scope of the Research



Scope of the Research



Applications of Cache Tuning

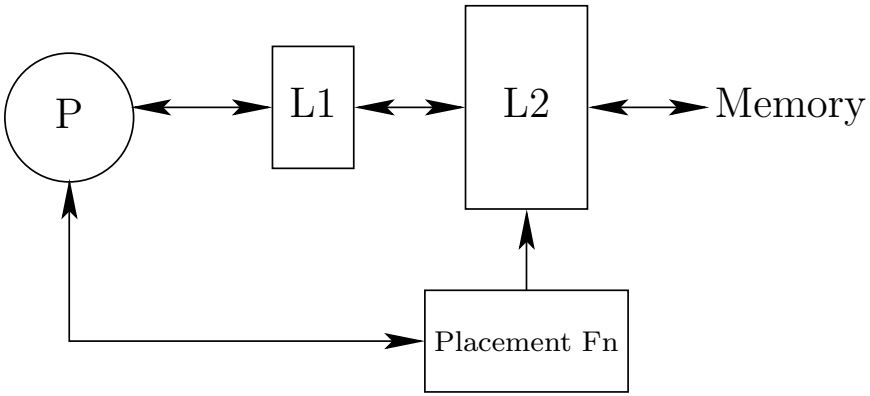
Where can tuning be applied?

- Scientific domain
- Embedded systems
- Fault tolerance

How can tuning be applied?

- As a one-time reconfiguration
- Statically at compile-time
- As a run-time optimization

Architecture Overview



Architecture Model

- Address translation complexity $\propto$ flexibility
- Gated-Vdd[a] technique used
- Programmable placement

[a] Powell '01

Implementation Overheads

- Area and energy costs are typically 1–2% of cache costs
- Latency costs can be masked
- Instruction set extensions

Software Driven Cache Tuning

Example 1

```
 1: a = const;
 2: b = const;
 3: c = const;
 4: placement(a, b, c)
 5: for i = 1 to N do
 6:     for j = 1 to N do
 7:         for k = 1 to N do
 8:             X[ai + bj + ck] + +;
 9:         end for
10:     end for
11: end for
```

Example 2

```
 1: c = const;
 2: for i = 1 to N do
 3:     a = expr;
 4:     for j = 1 to N do
 5:         b = expr;
 6:         placement(a, b, c)
 7:         for k = 1 to N do
 8:             X[ai + bj + ck] + +;
 9:         end for
10:     end for
11: end for
```

- Statically scheduled placement function adaptations are placed before specific program regions, e.g., nested loops

Optimizing for Scientific Computation

Strided placement

- Placement is customized to access stride
- Extensible to multiple arrays and multiple strides

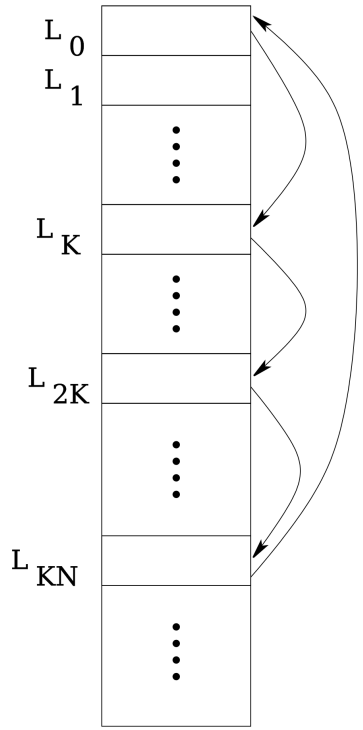
Construct conflict sets based on *stride*

- For performance $\longrightarrow$ spread accesses across conflict sets
- For energy savings $\longrightarrow$ fold accesses to fewer conflict sets
- Increasing associativity and/or cache size is a program agnostic way of construction conflict sets

Strided Placement

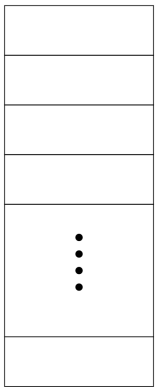
Access pattern: L_0, L_K, L_{2K}, ..., L_0, L_K, L_{2K},..... L_1, L_{K+1}, ...

Main memory



Modulo placement (S=K)
All elements of a stride set map to the same conflict set

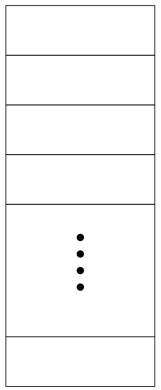
Direct mapped



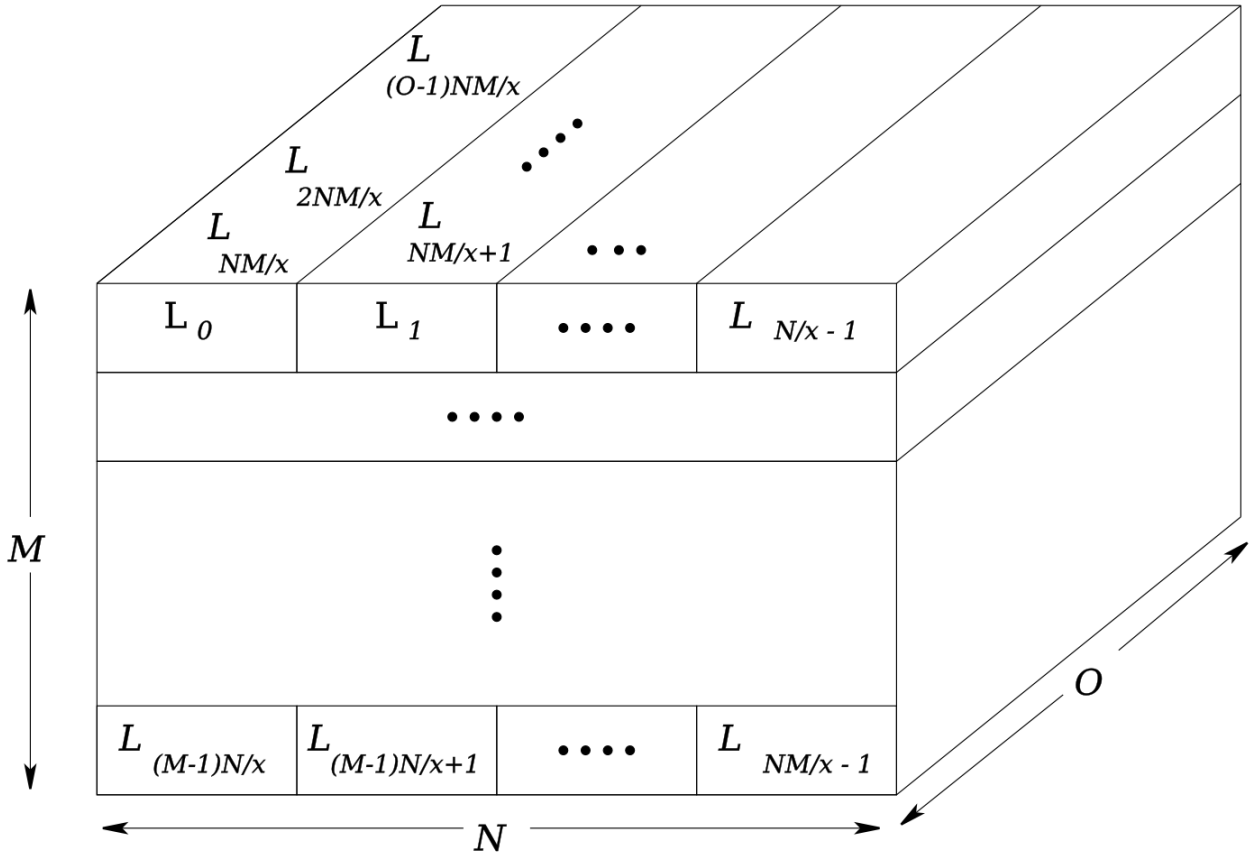
Fully associative

Strided placement
Elements of a stride set map to different conflict sets

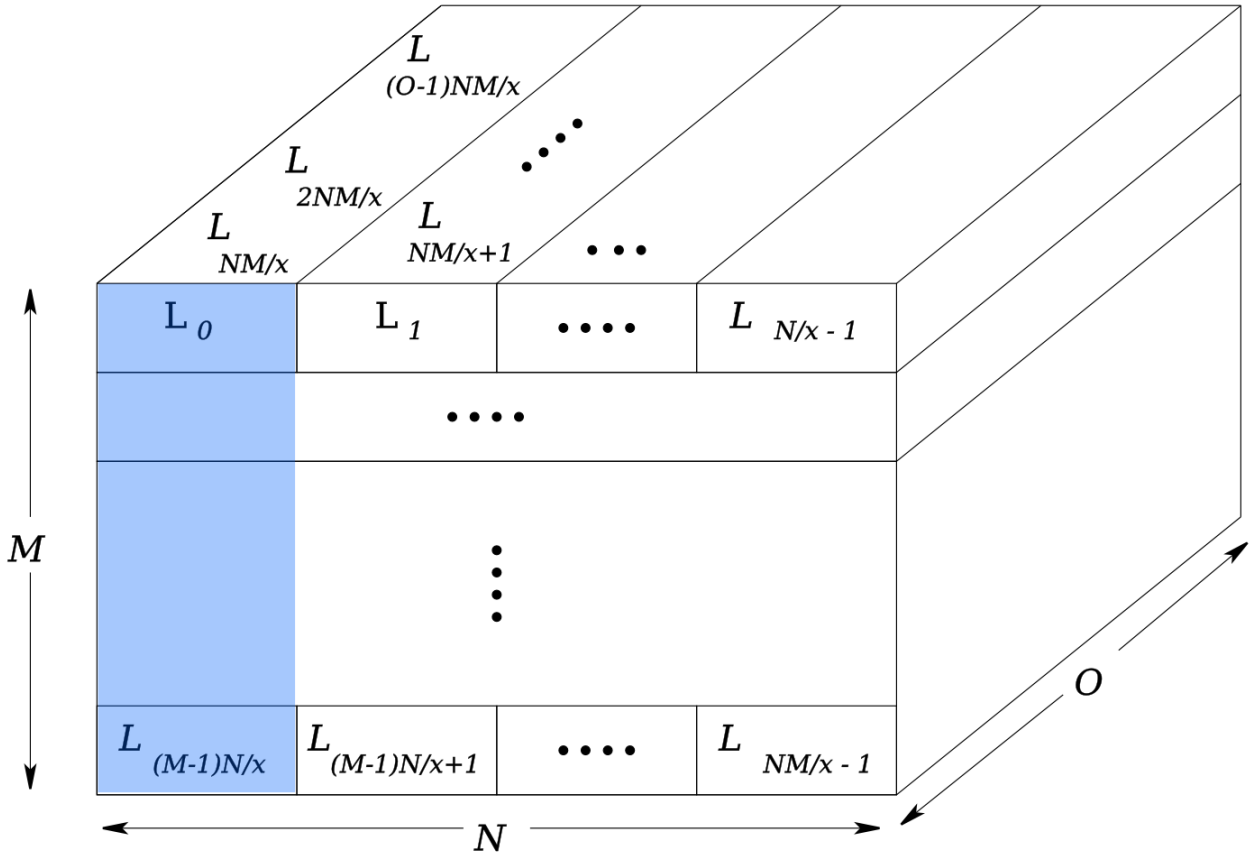
Direct mapped (S=K)



Conflict Set Construction

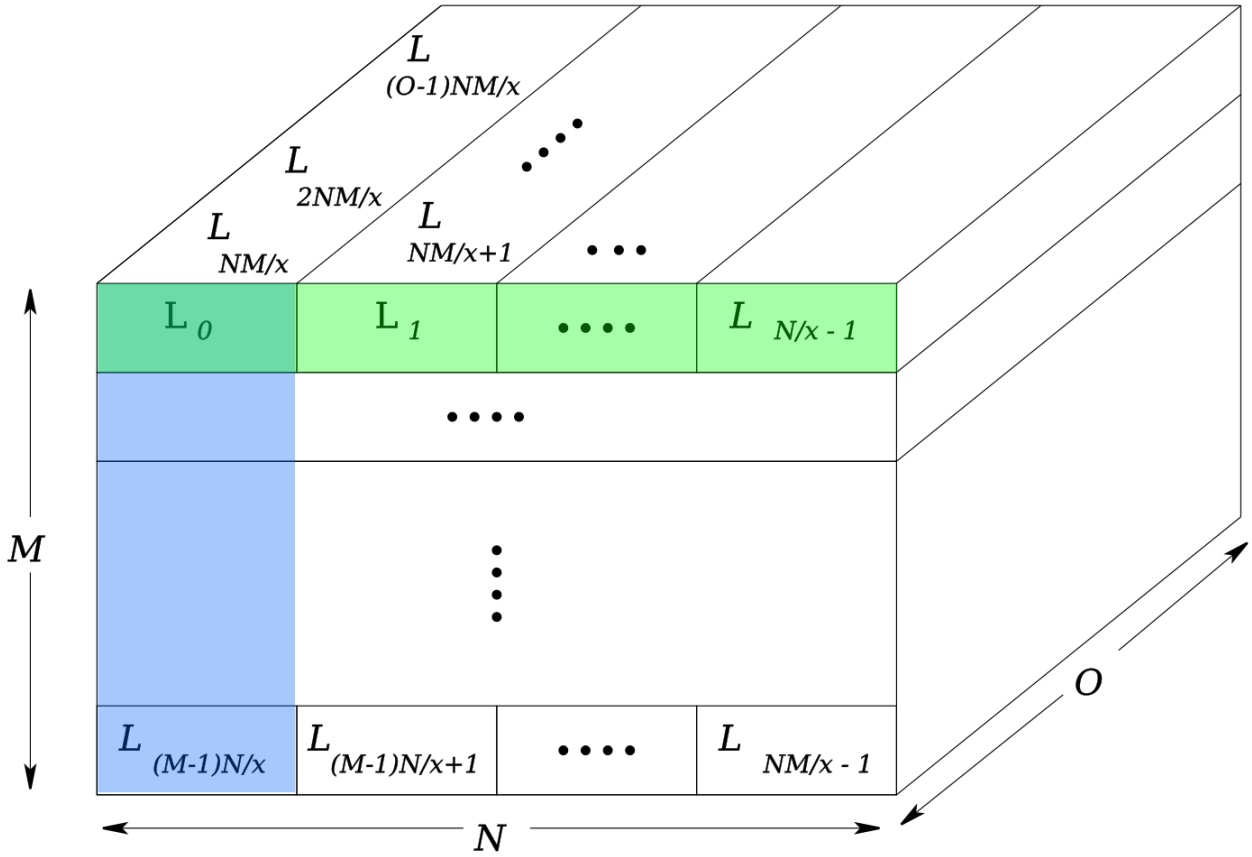


Conflict Set Construction



x-order access: Each column is a conflict set (L mod S)

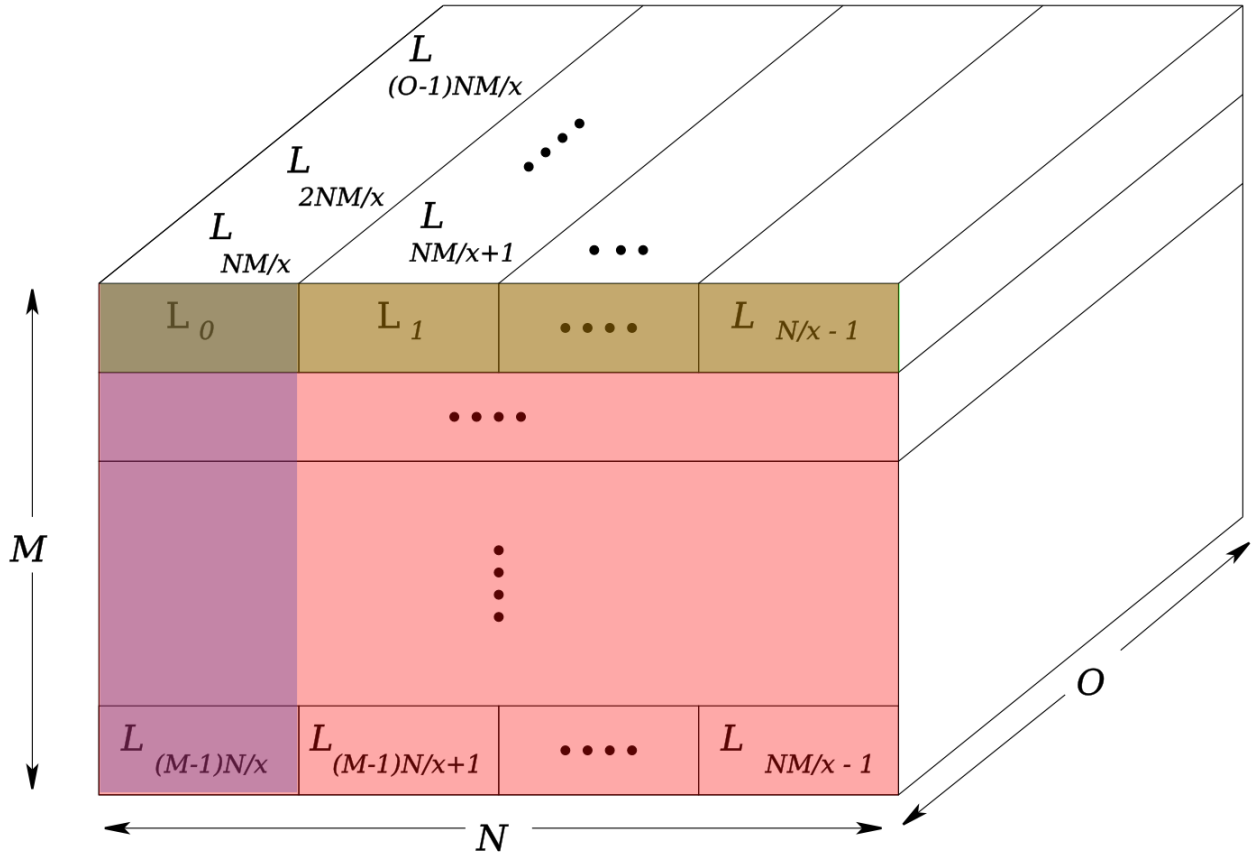
Conflict Set Construction



x-order access: Each column is a conflict set (L mod S)

y-order access: Each row is a conflict set (L /(N/x) mod S)

Conflict Set Construction



x-order access: Each column is a conflict set (L mod S)

y-order access: Each row is a conflict set (L /(N/x) mod S)

z-order Diagonal access: Each row is a conflict set (L /(NM/x) mod S)

Strided Placement: Algorithms

Shaping Algorithm

Require: l_i, K, sn_k, S, k_a
Ensure: $cache\_set(l_i)$

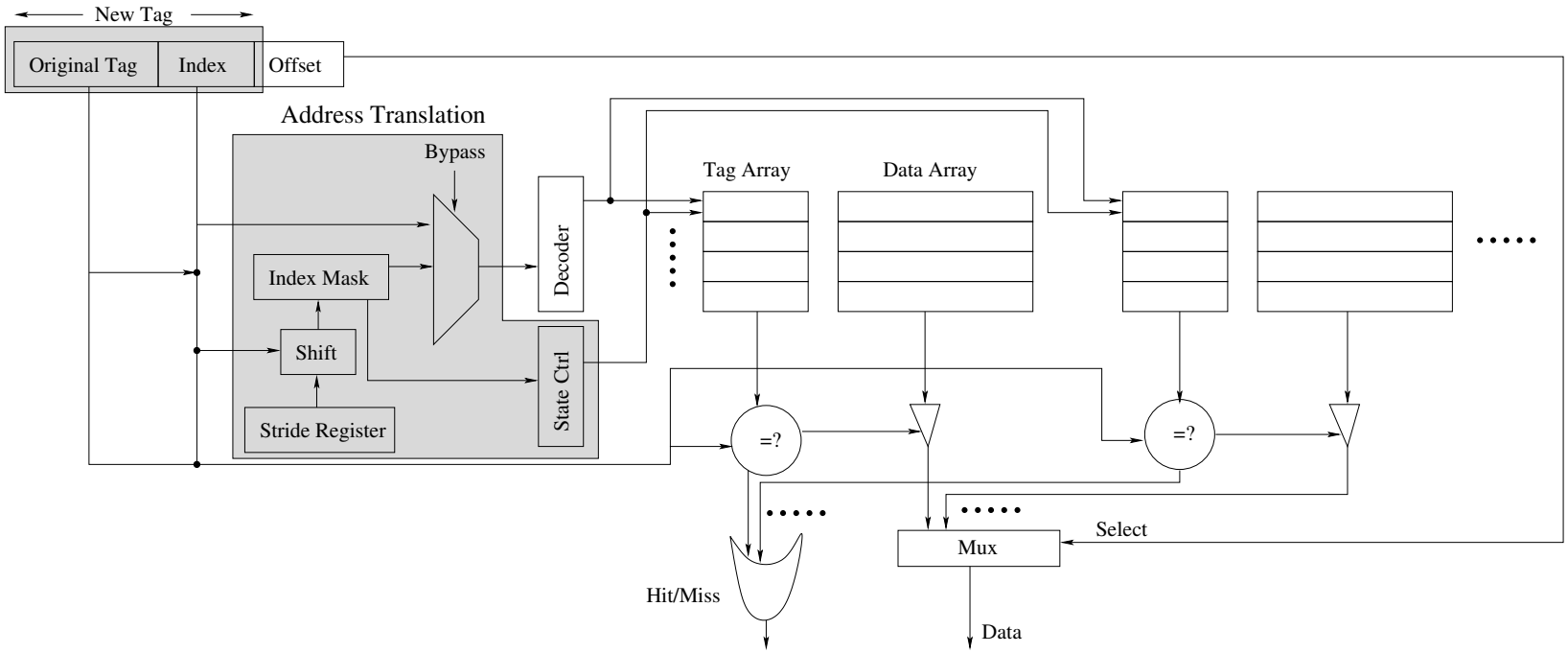
1: **if** $bypass = 1$ **then**
2: $return(l_i \ mod \ S)$ {Modulo placement}
3: **else**
4: $S_a = active\_sets\ (sn_k, S, k_a)$
5: $return\ (\frac{l_i}{K * k_a} \ mod \ S_a)$
6: **end if**

Sizing Algorithm

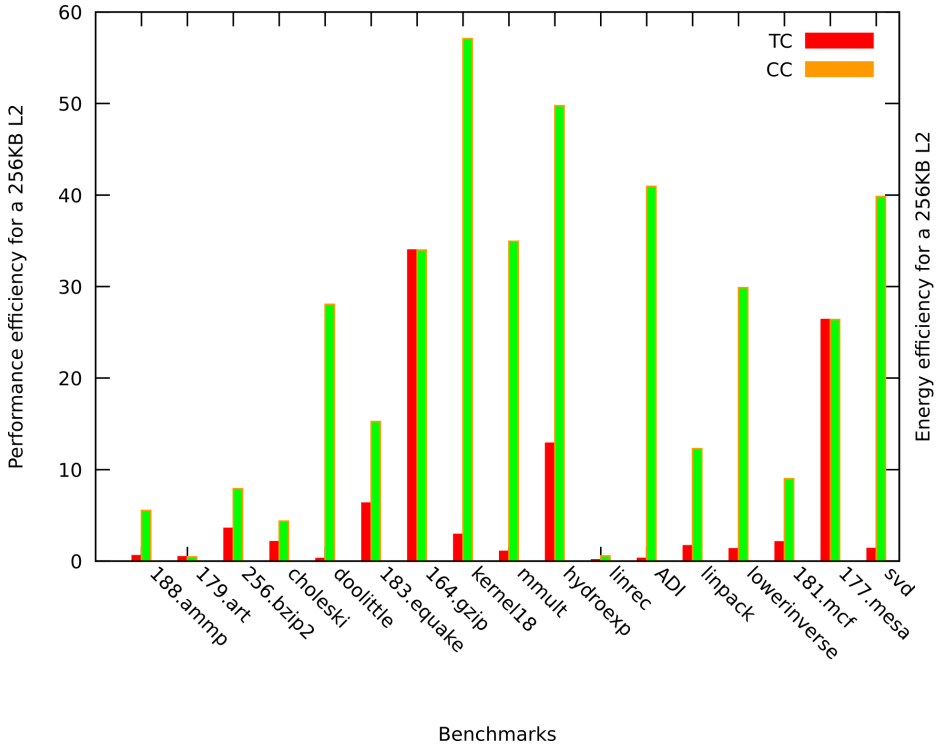
Require: sn_k, S, k_a
Ensure: S_a, the number of active cache sets

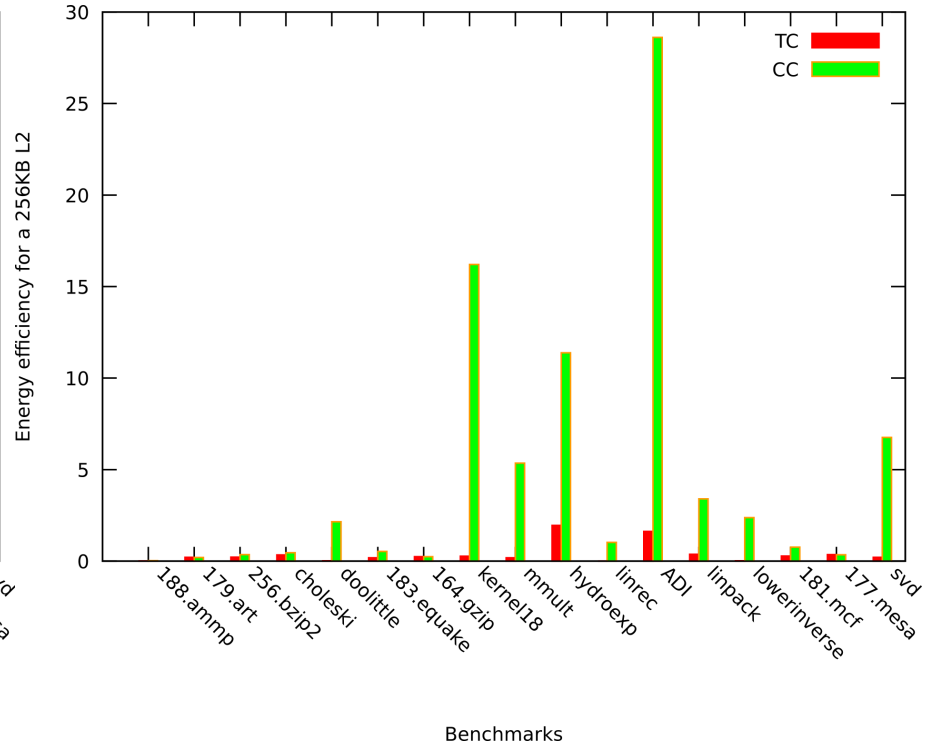
1: **if** $Sn_K < S * k_a$ **then**
2: $return\ (S)$ {Keep all sets active}
3: **else**
4: $return\ (S - \frac{sn_k}{k_a})$
5: **end if**

Strided Placement: Hardware Implementation



Results





- Significant improvements across the board

Dynamic Strategies: Cache Folding

Next step in evolution of cache energy optimizations

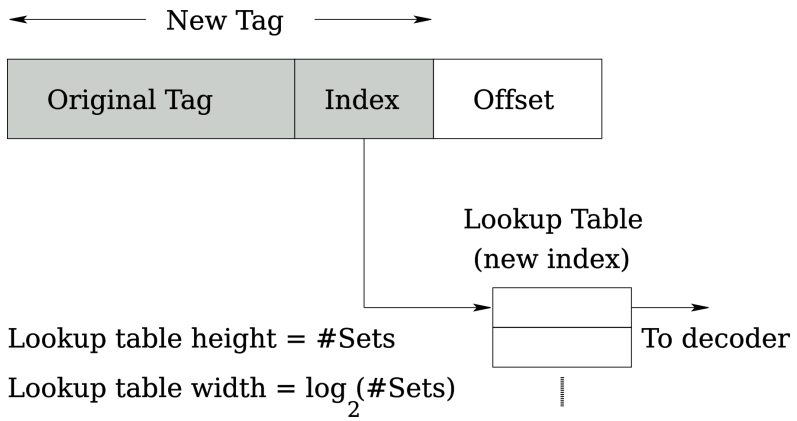
- Memory regions mapping to disjoint cache resources are combined to share cache sets enabling power off optimizations
- Caches are *scaled* $\rightarrow$ segments turned off will not be accessed
- Non-uniform scaling $\rightarrow$ some memory lines get relatively lower cache resources allocated

Folding Heuristics

- Decay resizing[a]
- Power of two resizing
- Segment resizing

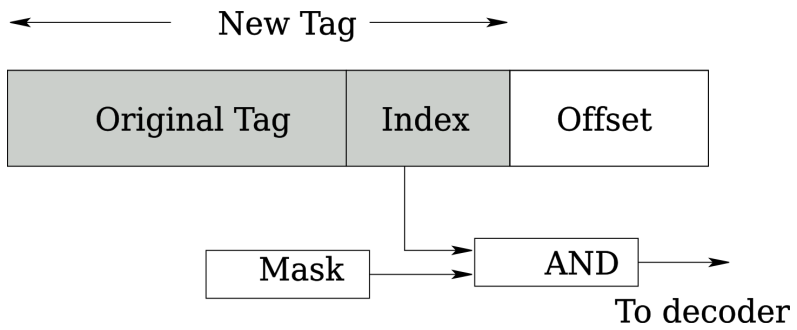
[a] Kaxiras '01

Folding Heuristics: Address Translation



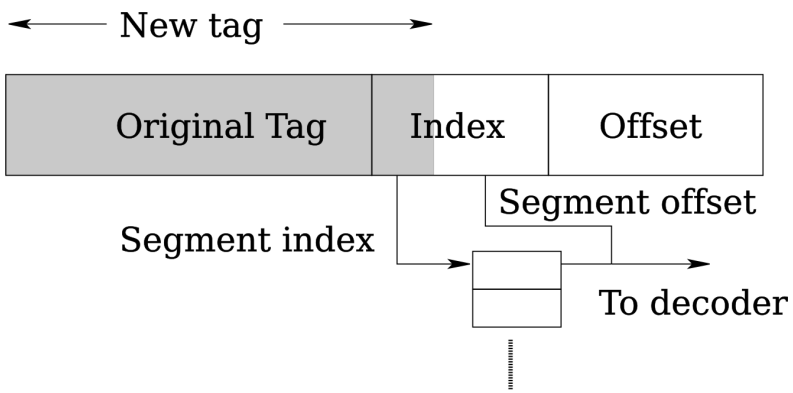
Decay resizing

Folding Heuristics: Address Translation



Power of two resizing

Folding Heuristics: Address Translation

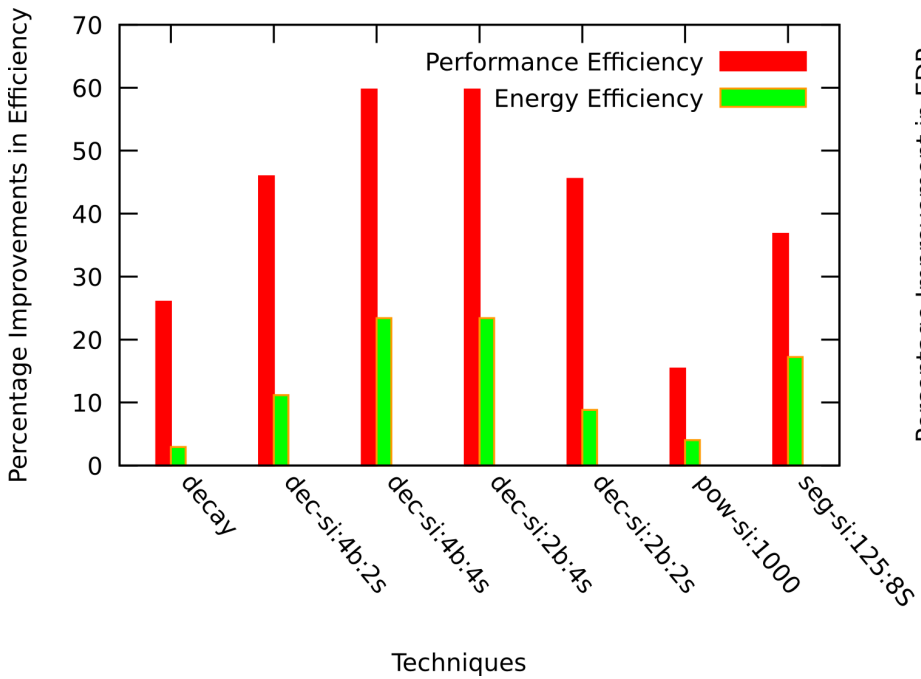


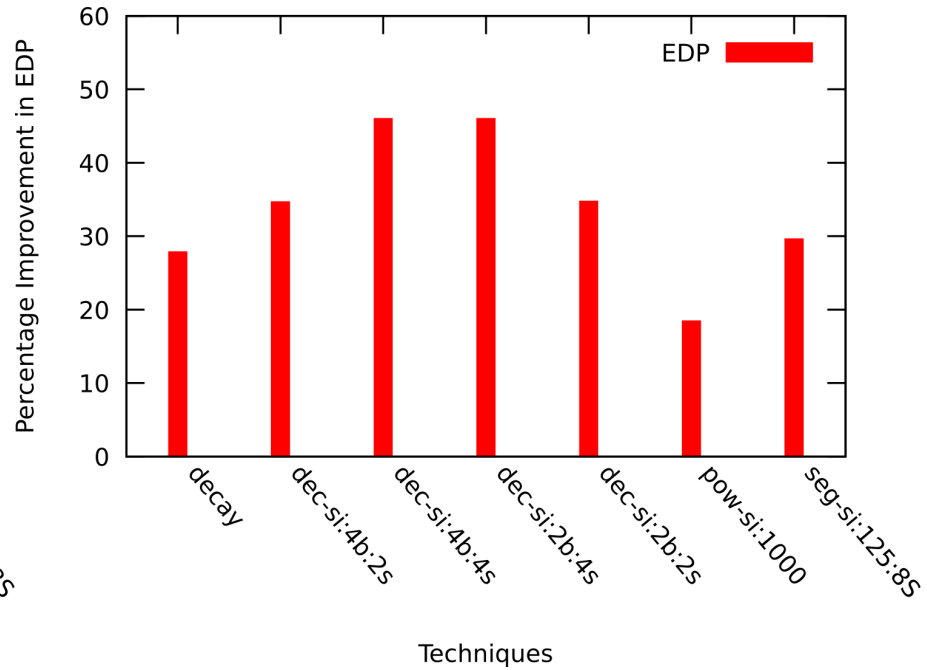
Lookup table height = #Segments

Lookup table width = $\log_2$ (#Segments)

Segment resizing

Efficiency Improvements using Folding





- Folding allows aggressively turning off cache lines to save leakage energy

Improving Variation Tolerance

SRAM Failure Modes

- Access time failures (too slow)
- Read failures (modifies or loses data)
- Write failures (data not modified)

Improving Variation Tolerance

SRAM Failure Modes

- Access time failures (too slow)
- Read failures (modifies or loses data)
- Write failures (data not modified)

Estimates place defect-free cache yield at 33%[a]

[a] Agarwal '04

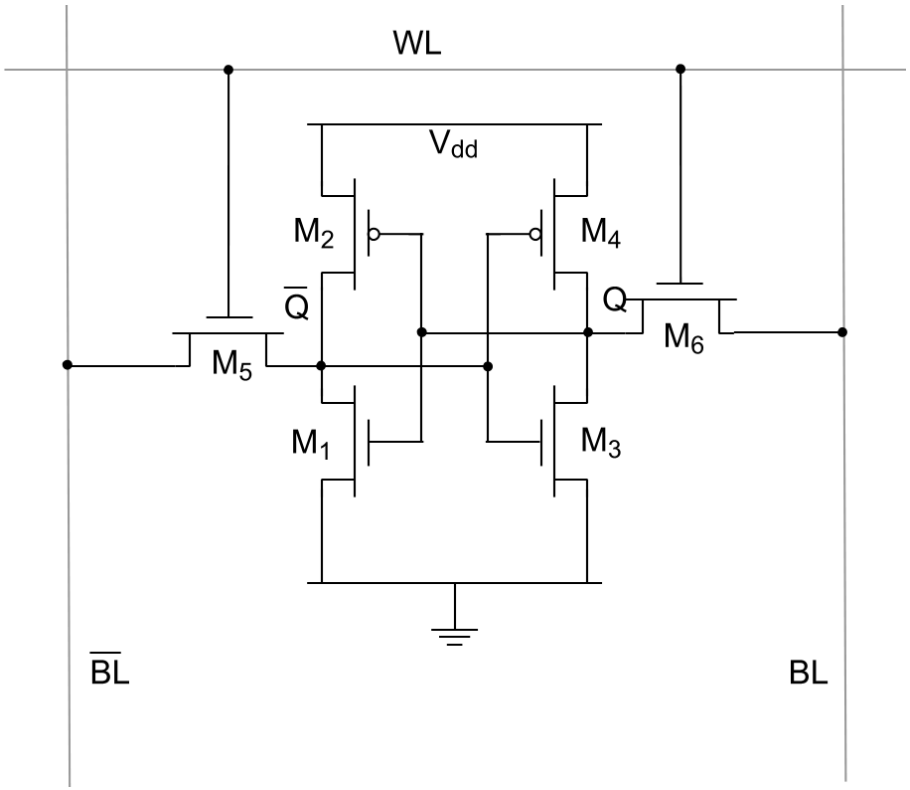
Improving Variation Tolerance

SRAM Failure Modes

- Access time failures (too slow)
- Read failures (modifies or loses data)
- Write failures (data not modified)

Estimates place defect-free cache yield at 33%[a]

[a] Agarwal '04



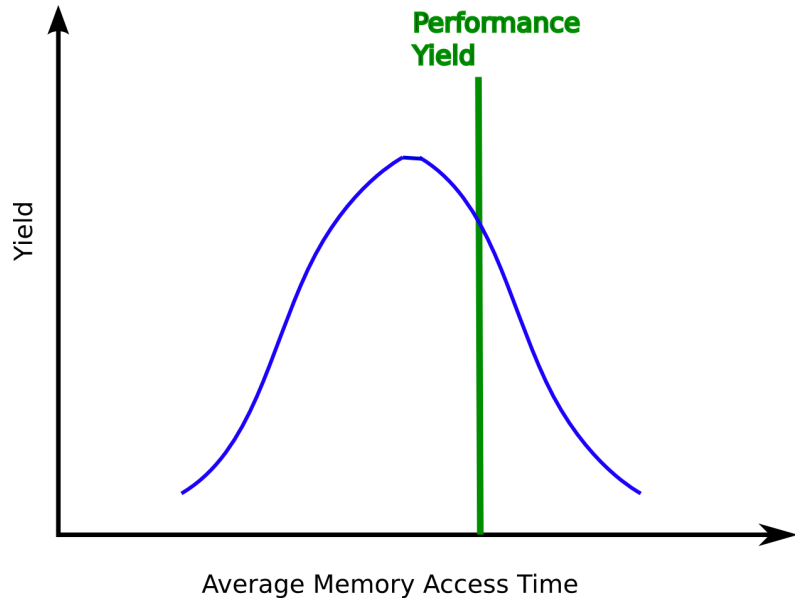
Impact of Process Variations on Cache Operation

Problem: Faulty Cache Lines

- Solution: Agnostic remap (e.g., to neighboring cache lines)
- Performance impact $\longrightarrow$ increase in misses

Cache Tuning for Fault Tolerance

- Improved sharing using **reference profiles** $\longrightarrow$ **shape** the cache
- Improve performance yield
- AMAT distribution narrows



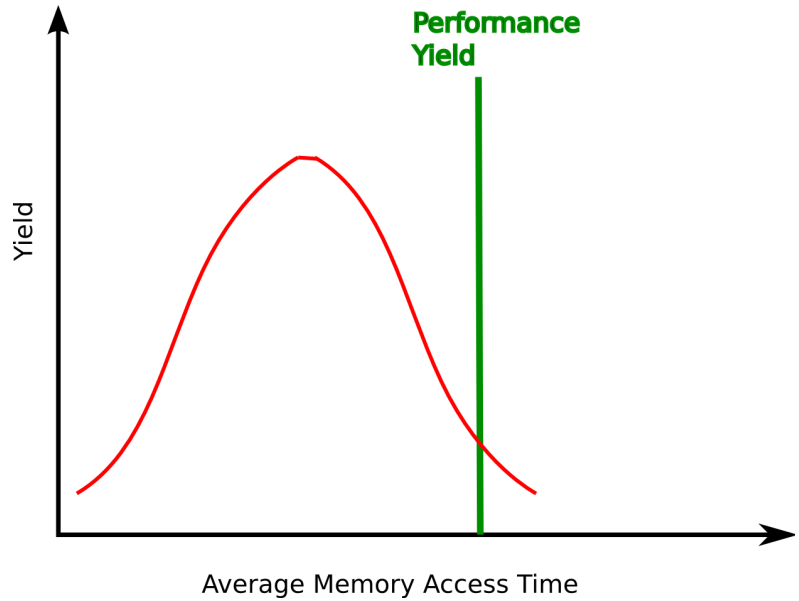
Impact of Process Variations on Cache Operation

Problem: Faulty Cache Lines

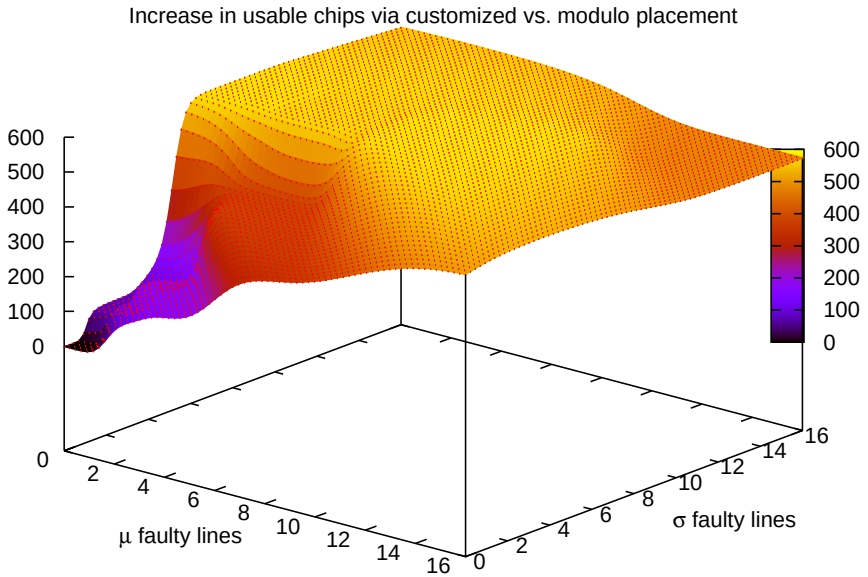
- Solution: Agnostic remap (e.g., to neighboring cache lines)
- Performance impact $\longrightarrow$ increase in misses

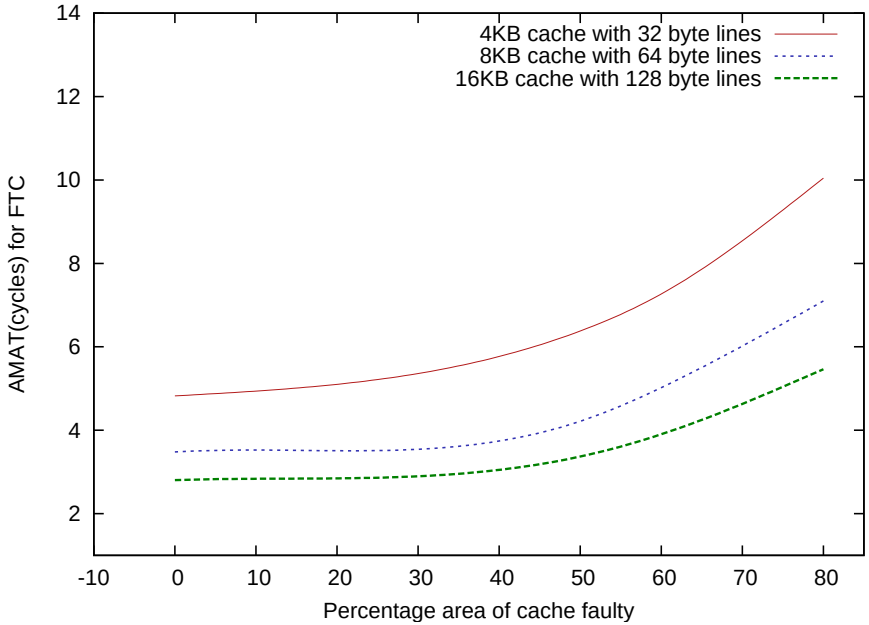
Cache Tuning for Fault Tolerance

- Improved sharing using **reference profiles** $\longrightarrow$ **shape** the cache
- Improve performance yield
- AMAT distribution narrows



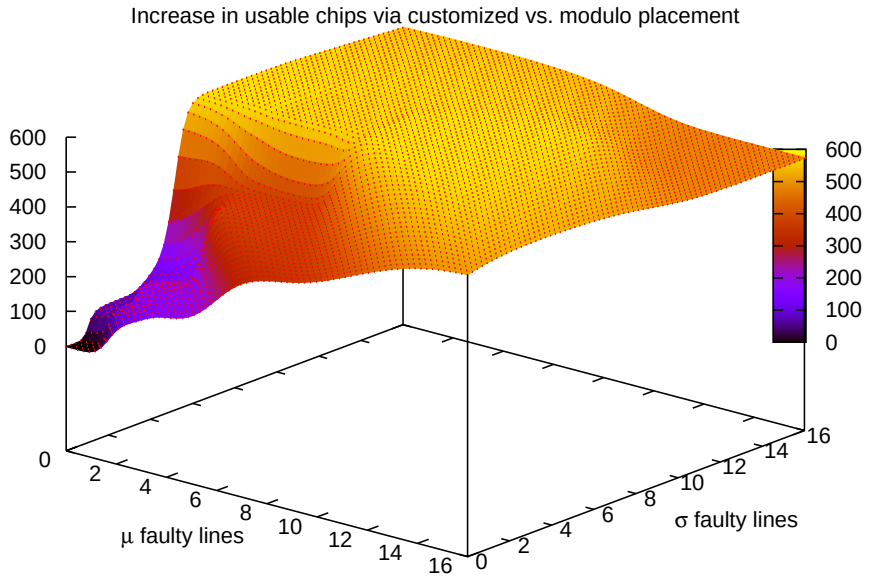
Improving Variation Tolerance

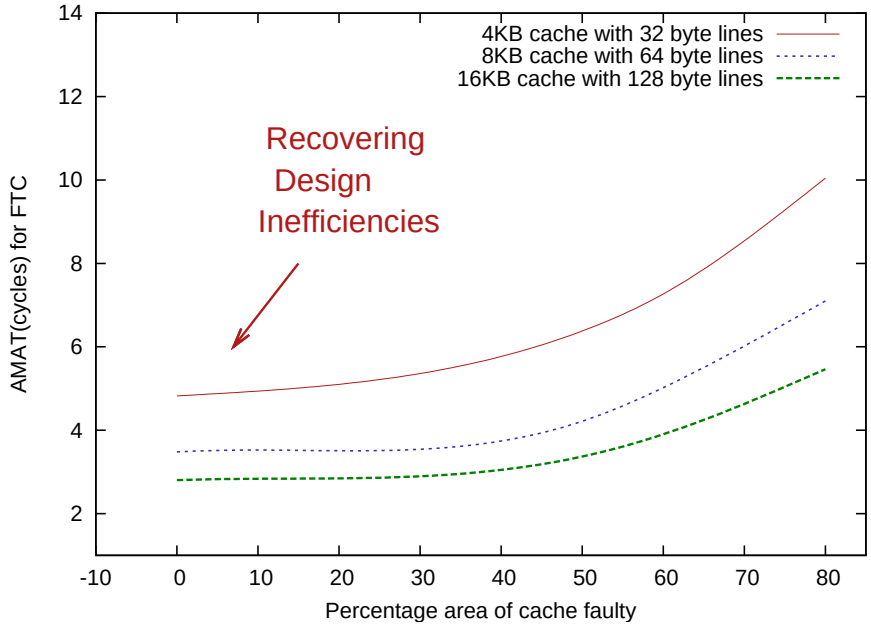




- **Performance yield** increases by up to four times over modulo placement strategies
- Cache redundancies exploited $\longrightarrow$ application to power management

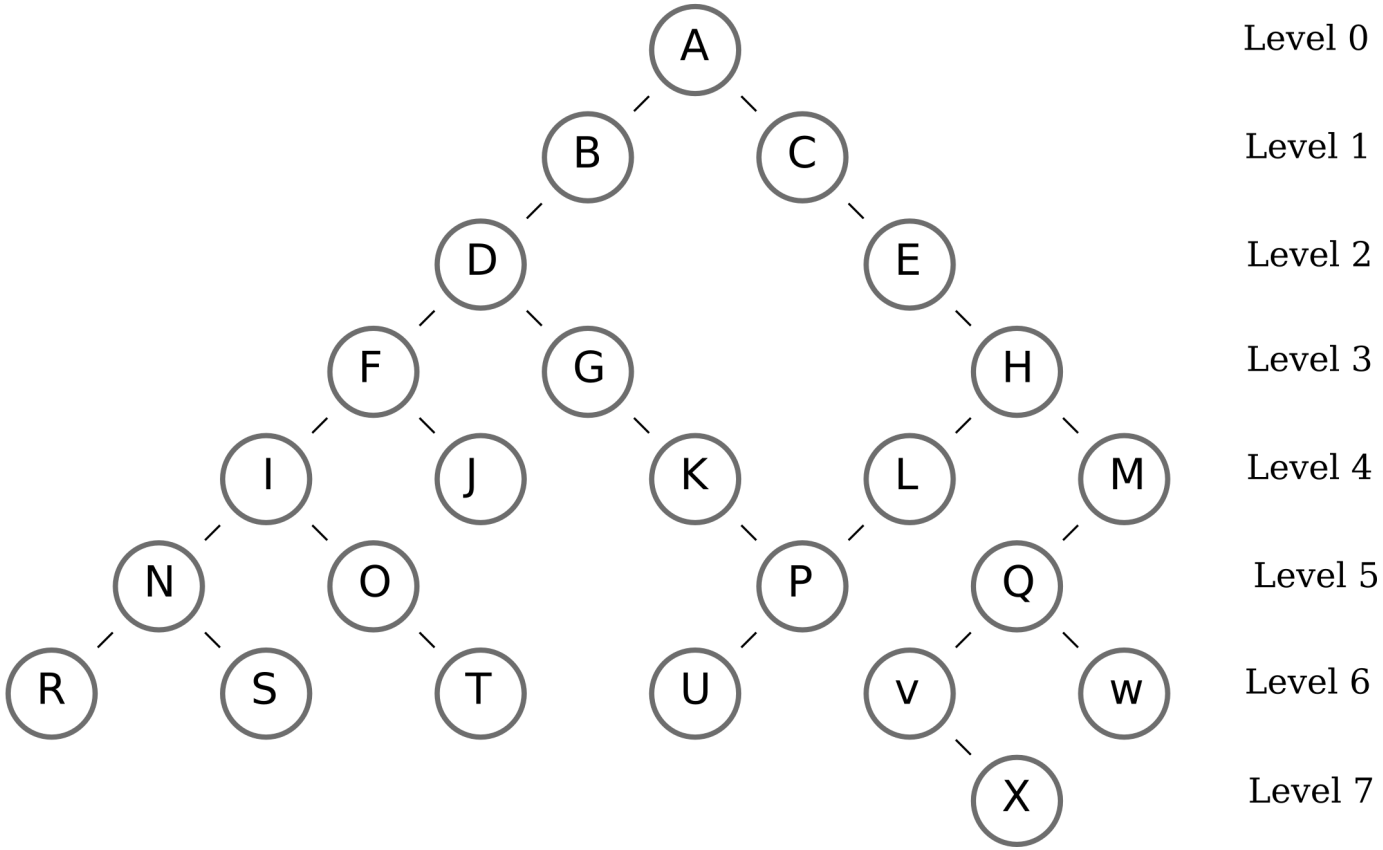
Improving Variation Tolerance



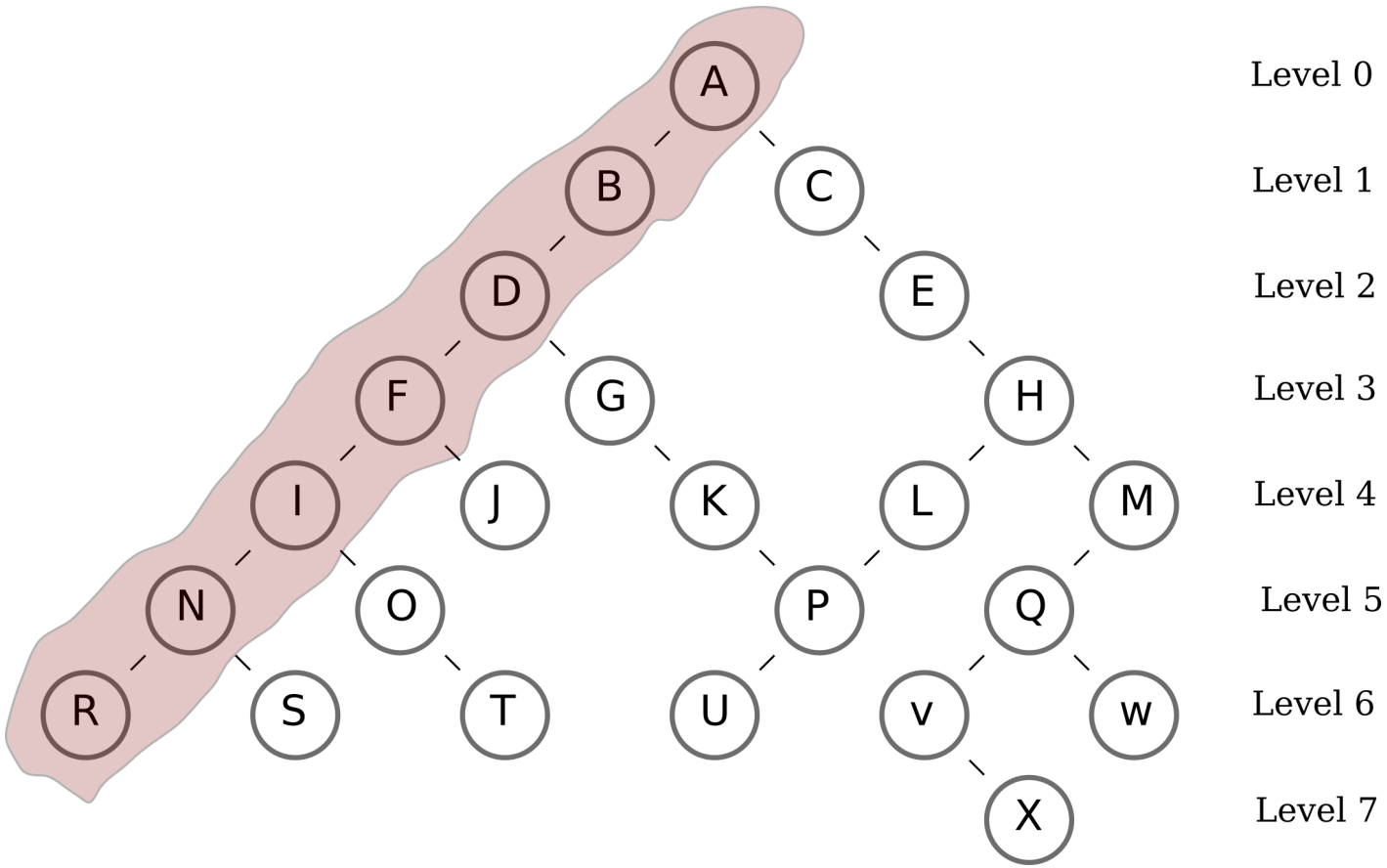


- **Performance yield** increases by up to four times over modulo placement strategies
- Cache redundancies exploited $\longrightarrow$ application to power management

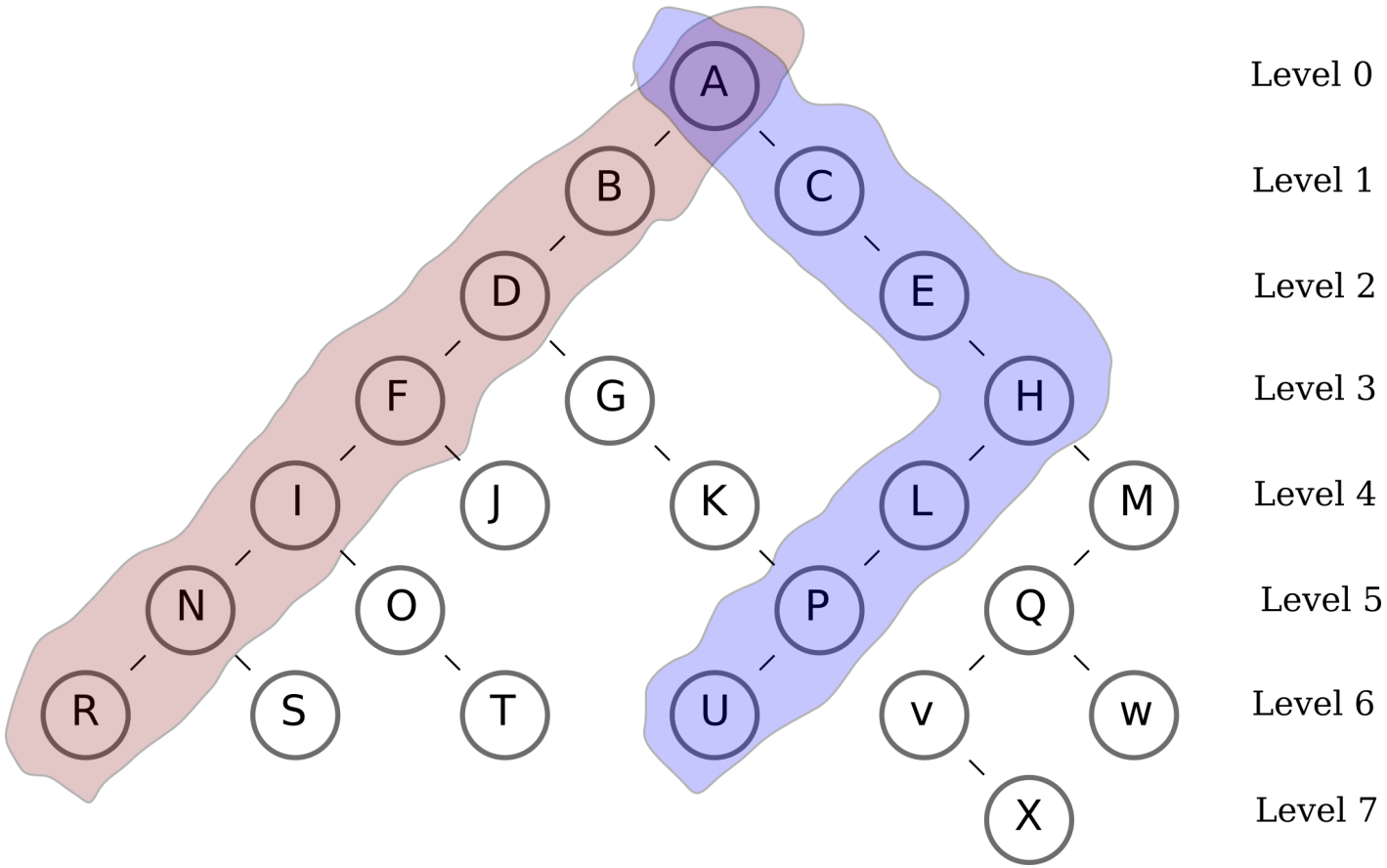
Domain Specific Optimizations: Data Trace Cache



Domain Specific Optimizations: Data Trace Cache

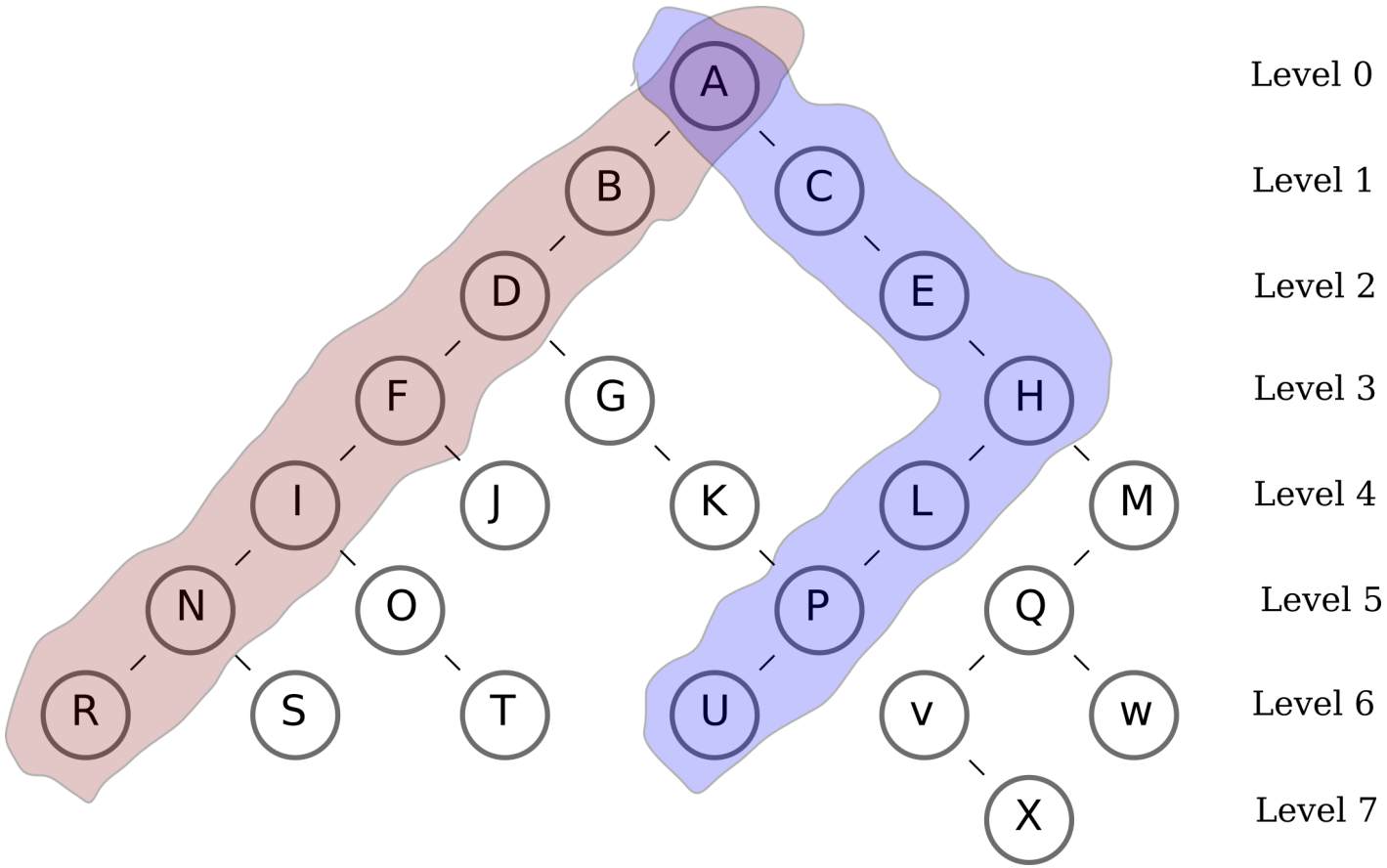


Domain Specific Optimizations: Data Trace Cache



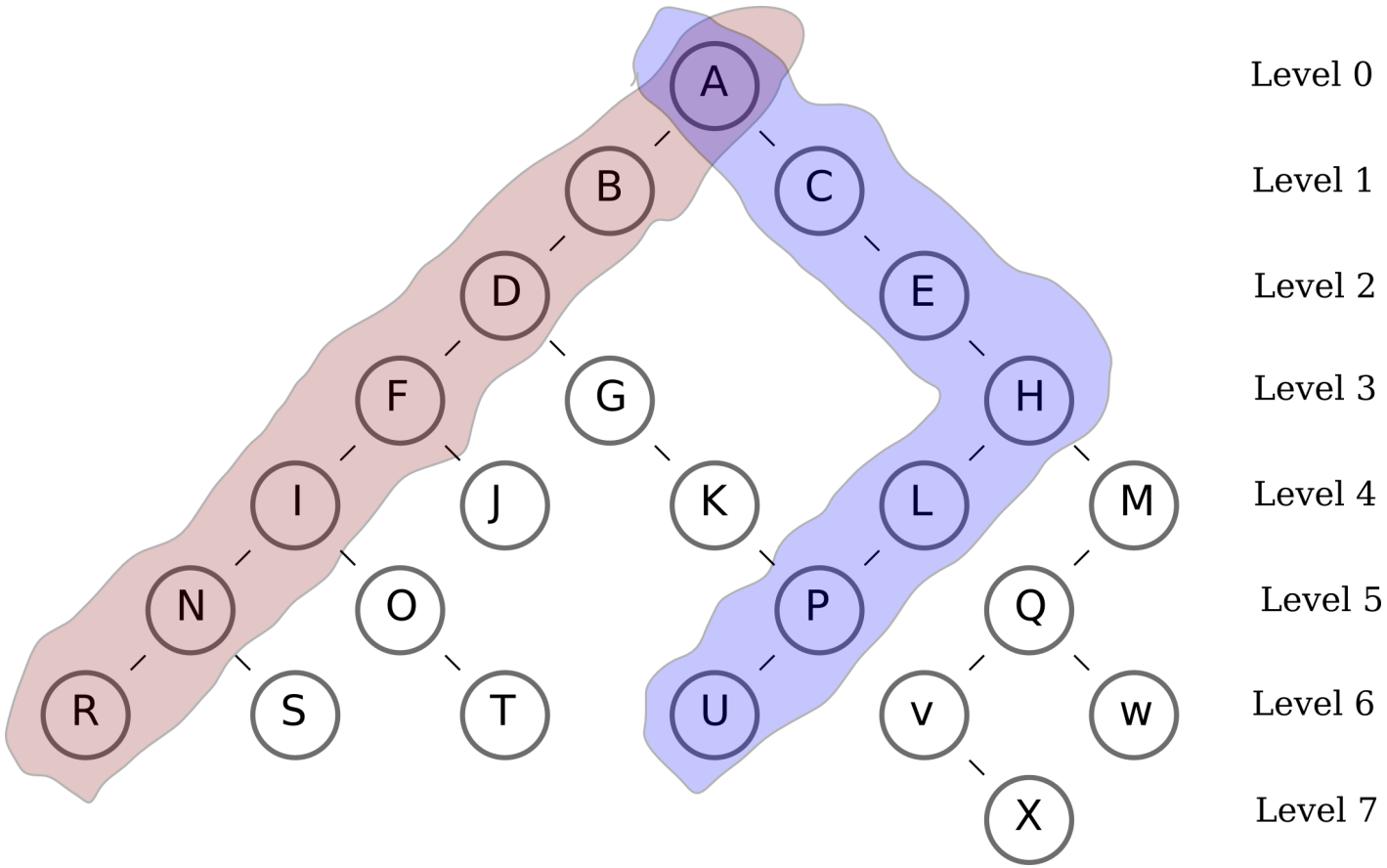
- Tree nodes conflict with leaf nodes

Domain Specific Optimizations: Data Trace Cache



- Tree nodes conflict with leaf nodes
- Entire traces need to be stored

Domain Specific Optimizations: Data Trace Cache



- Tree nodes conflict with leaf nodes
- Entire traces need to be stored
- **Make each level a conflict set group!**

Extensions to Multithreading

- Efficiency and utilization for multi-threaded programs are low

Extensions to Multithreading

- Efficiency and utilization for multi-threaded programs are low
  - Threads and processes stomp on each other

Extensions to Multithreading

- Efficiency and utilization for multi-threaded programs are low
  - Threads and processes stomp on each other
- Cache tuning can shape the cache to mitigate interference and increase data reuse $\rightarrow$ increased efficiency

Utilization based tuning

- Use utilization values across program execution to size caches

Utilization based tuning

- Use utilization values across program execution to size caches
- Fold conflict sets on individual utilizations

Utilization based tuning

- Use utilization values across program execution to size caches
- Fold conflict sets on individual utilizations
- Identify non-overlapping live ranges for folding

Voltage Scaled Caches

- Scale supply voltages in caches for energy savings $\rightarrow$ more defects

Voltage Scaled Caches

- Scale supply voltages in caches for energy savings $\rightarrow$ more defects
- Increase in defects $\rightarrow$ performance degradation

Voltage Scaled Caches

- Scale supply voltages in caches for energy savings $\rightarrow$ more defects
- Increase in defects $\rightarrow$ performance degradation
- Ameliorate effects of performance degradation by tuning the cache

Related Work in Improving Cache Performance

Related Work in Improving Cache Performance

Hardware Strategies

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

- Data relayout

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

- Data relayout
- Data pre-fetching

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

- Data relayout
- Data pre-fetching
- Scheduling optimizations

Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

- Data relayout
- Data pre-fetching
- Scheduling optimizations
- Scratchpad memories

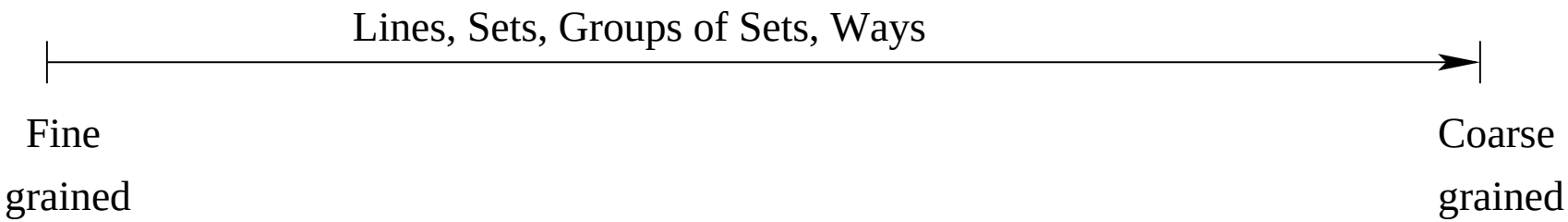
Related Work in Improving Cache Performance

Hardware Strategies

- Pseudo-associativity
- Indexing schemes
- Cache replacement optimizations
- Partitioned caches and NUCA caches
- ECC, Redundancy and Neighbor remapping

Software Strategies

- Data relayout
- Data pre-fetching
- Scheduling optimizations
- Scratchpad memories



Summary and Future Opportunities

Recap

- Caches consume significant resources, but are inefficient
- As designs evolve, the inefficiency of caches will bear on costs
- Design efficient caches by managing cache resources to benefit program performance and save energy
- Cache tuning offers significantly increased efficiency with moderate costs
- Cache tuning is applicable to various domains and requirements

Future Extensions

- Applicability to multi-core domains $\longrightarrow$ private, shared, inclusive, exclusive caching
- Advanced compiler optimizations made possible

Publications

- From Adaptive to Self-tuned Systems (*Book Chapter, The Future of Computing, '07*)
- Improving Cache Efficiency via Resizing + Remapping (*ICCD '07*)
- Customized Placement for High Performance Embedded Processors (*ARCS '07, LNCS Vol. 4415 '07*)
- Adaptive Cache Placement for Scientific Computation (*Tech Report - GIT-CERCS-07-03 '07*)
- Customizable Fault Tolerant Caches for Embedded Processors (*ICCD '06*)
- Data Trace Cache: An Application Specific Cache Architecture (*MEDEA '05, SIGARCH CAN March '06*)

Strided Placement: Adaptations from Data Skewing

Analogy to memory skewing

Caches $\longrightarrow$ memory banks

Arrays $\longrightarrow$ memory lines

Multiple Strides and Multiple Arrays

- One array, two strides (p, q) within a loop $\longrightarrow$ optimize placement for stride $K^{=} gcd(p, q)$
- Two arrays different strides (p, q) within a loop $\longrightarrow$ optimize placement for stride $K^{=} gcd(p, q)$
- Associativity and cache sizes also play an important role

Relation to Layout Optimizations

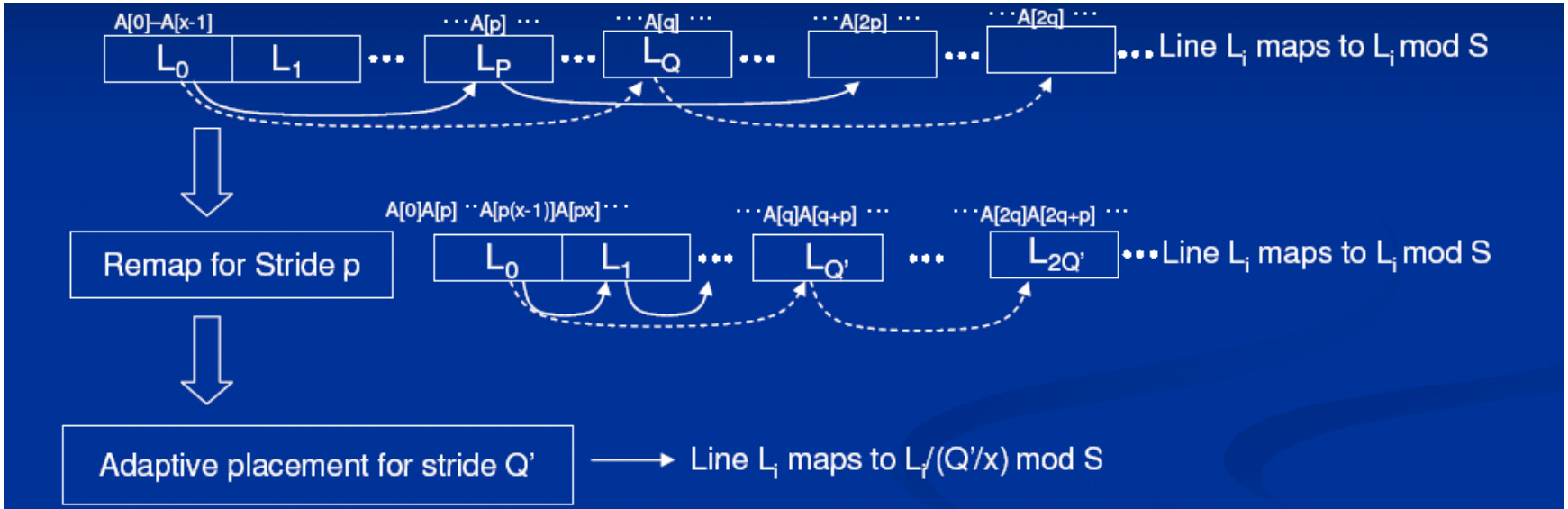
The proposed technique does not alter the data layout, physical or virtual, in any manner, and is therefore complementary to existing compiler optimizations

Relation to Layout Optimizations

The proposed technique does not alter the data layout, physical or virtual, in any manner, and is therefore complementary to existing compiler optimizations

Newer more powerful compiler optimizations can be formed with compilers tuning the cache as well as optimizing for the cache: blocking, remapping etc.

Leveraging Compiler Optimizations:Example



- Customized placement does not alter the memory layout or memory access pattern
  - Complementary to re-mapping, pre-fetching, loop transformations, etc.
  - Can be driven by run-time stride detection

Fault Tolerant Cache: Simulation Methodology

Simulation model and setup

- Embedded processor model with single cycle cache access time and single level cache
- DRAM latency assumed 100 cycles
- Assume cache has BIST circuitry to identify defects post-fab
- Tools used included *Valgrind* and *CACTI 3.0*
- Simulations performed on *Mibench* benchmarks

Active Management Costs

- Area increase is 2–5%
- Switching energy per access increases by 5–8%
- Latency impact is 0.25ns at 90nm $\rightarrow$ processors below 1Ghz can absorb this penalty in a single cycle

Overhead: Writebacks

- Eager writebacks can decrease execution time (Lee '00)
- Writebacks when applied during cache set turn-offs can increase performance (Kaxiras '01)
- Writes are a small fraction of accesses ($< 5\%$)

Strided Placement: Simulation Methodology

Simulation model and setup

- 15 cycle L2 access times
- DRAM latency assumed 200 cycles
- Tools used included *SimpleScalar* and *CACTI 3.0*
- Simulations performed on *Spec2000*, *Linpack*, *Livermore*, *Linear Algebra* programs

Active Management Costs

- Added additional latency of one cycles for lookup
- Area and energy costs $< 1\%$
- Programming placement adds ≈ 5000 cycles $< 1\%$ of total execution time
- Writebacks required

Dynamic Folding: Simulation Methodology

Simulation model and setup

- 15 cycle L2 access times
- DRAM latency assumed 200 cycles
- Tools used included *SimpleScalar* and *CACTI 3.0*
- Simulations performed on *Spec2000*, *DIS*, *Olden* benchmarks

Active Management Costs

- Added additional latency of one cycles for lookup
- Area and energy costs $< 1\%$
- Programming placement adds ≈ 5000 cycles $< 1\%$ of total execution time
- Writebacks required

Data Trace Cache: Simulation Methodology

Simulation model and setup

- Embedded processor model with single cycle cache access time and single level cache
- DRAM latency assumed 100 cycles
- Tools used included *Valgrind* and *CACTI 3.0*
- Simulations performed on *Mibench* benchmarks

Active Management Costs

- Added latency of two cycles for lookup
- Area increase is 5% (lookup table)
- AMATs lowered by $\approx$ 40%

The Synonym Problem

- Consider an eight line memory, two set cache
  - L_0 (Address:000, Tag:00) mapped to cache set CS_0 with modulo placement
  - L_1 (Address:001, Tag:00) mapped to cache set CS_1 with modulo placement
  - L_0 (Address:000, Tag:00) re-mapped to cache set CS_1 with new placement
  - Access on L_0 results in a false hit as tag matches that of L_1

Dealing with Synonyms

- Use an extended tag $\longrightarrow$ tag bits appended with unmodified cache set address $\longrightarrow$ similar to appending PIDs in virtual memories
  - L_0 (Address:000, Tag:000) re-mapped to cache set CS_1 with new placement
  - L_1 (Address:001, Tag:001) mapped to cache set CS_1 with modulo placement
  - Access to L_0 (Address:000, Tag:000) fails tag match as L_0 and L_1 have different tags

Related Work

- Memory skewing (Chang, Kuck and Lawrie '77, Raghavan and Hayes '93)
- Cache decay and generational behavior (Kaxiras '01)
- Declining effectiveness of dynamic caching (Burger '95)
- Dominance of leakage current (Kim '05, Borkar '99, Ranganathan '00, Zhang '02)

Related Work: Software Strategies

- Relayout (Rabbah '03, Chilimb '99, Panda '01)
- Loop transformations (Ghosh '99)
- Scratchpad memories (Panda '97, Steinke '02, Udayakumaran '06)
- Prefetching (Luk and Mowry '99)

Related Work: Hardware Strategies

- Pseudo-associativity (Jouppi '90, Qureshi '05, Hallnor '00, Agarwal '93)
- Indexing and Hashing (Seznec '93, Kharbutli '04, Zhang '06)
- Replacement (Jiang '02, Subramanian '06)
- Compression (Alameldeen '04, Zhang '00)
- Partitioned caches (Lee '00, Qureshi '05, Chang '06)
- NUCA caches (Kim '02, Chishti '03)
- Memory Controllers (Carter '99)

Related Work: Variation Tolerance in Caches

- Miss on accesses to faulty lines (Patterson '83)
- Add ECC (Kalter '90)
- Add redundancy (Turgeon '91, Nokolos '95)
- Map faulty blocks to neighbors (Shirvani '99, Agarwal '04)

Related Work: Energy Savings

- Turning off ways based on offline analysis (Albonesi '99)
- DRI-cache (Powell '01)
- Cache decay (Kaxiras '01)
- Drowsy cache (Flautner '02)
- Miss tags based resizing (Zhang '02)
- Adaptive mode control (Zhou '01)
- IATAC (Abella '05)

Machine Configuration

- Superscalar out of order processor - four instrn/cycle issue
- 15 cycle L2 latency, 2 cycle L1 latency
- Bus width 8 bytes

Placement Algorithm for FTC

Require: $r[S][W], f, ip[S][S]$
Ensure: $map[S]$

1: **for** $iter = 0$ to $S - 1$ **do**
2: $map[iter] = iter$ {Initialize}
3: **end for**
4: **for** $iter = 1$ to f **do**
5: find i, j s.t. $ip[i][j] = min(ip[S][S])$ {Find conflict sets having minimum ip}
6: $map[j] = map[i]$ {Merge the two conflict sets}
7: $update(r[S][W])$ {Update reference counts}
8: $update(ip[S][S])$ {Update the ip matrix}
9: **end for**
10: $update(map[S])$ {Update to remove mapping to faulty lines}