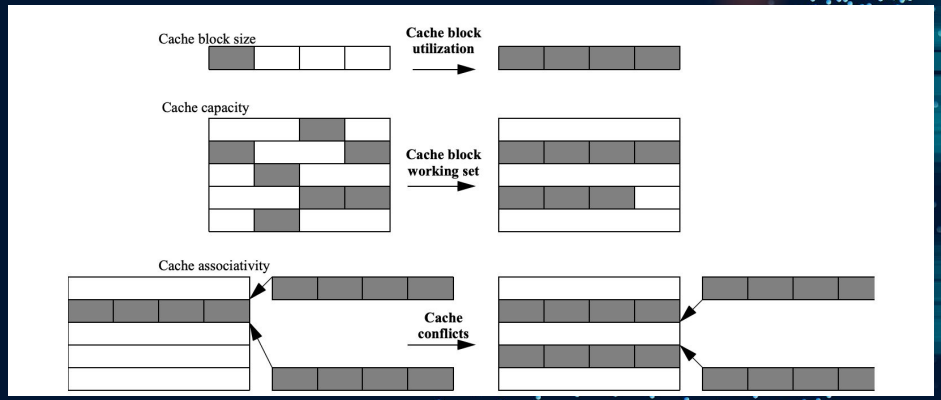
Cache-Conscious Structure Definition

Manav Khanvilkar & Ryan Jain

Group 30

Problem: Caches vs Data Structures

- Modern CPUs are fast; memory is slow → **cache gap**
- Data structures used in code are not optimized for **spatial** and **temporal locality**
- Modern code is often **pointer-heavy,** so performance is limited by cache misses

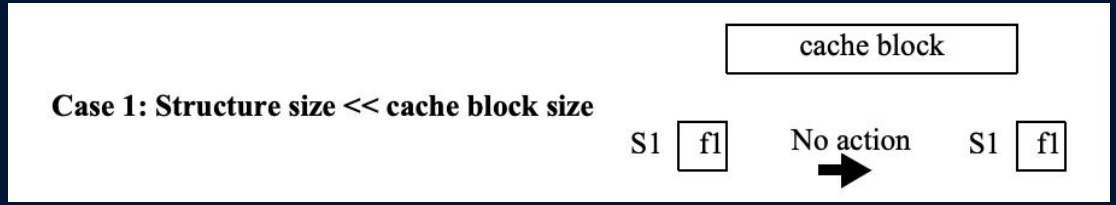


Background: Memory Hierarchy & Cache Metrics

- Latency gap: CPU → L1 → L2 → Memory
- Cache blocks: fixed-size units (16–64 bytes)
- Key metrics:
  - Cache block utilization
  - Working set / cache pressure
  - Cache miss rate

Background: Existing Cache-Conscious Layout

- Prior work focuses on object placement, not field layout
- Chilimbi & Larus: co-locate related objects
- Effective for small objects (< cache block)
- Falls short for larger structures



Motivation: Three Structure Size Regimes

- Small structures (< cache block)
  - Entire object fits in one block → great locality
- Medium structures (~1–2 blocks)
  - Hot/cold fields mix → **splitting** improves locality
- Large structures (≫ cache block)
  - Many fields per object → **field reordering** improves locality

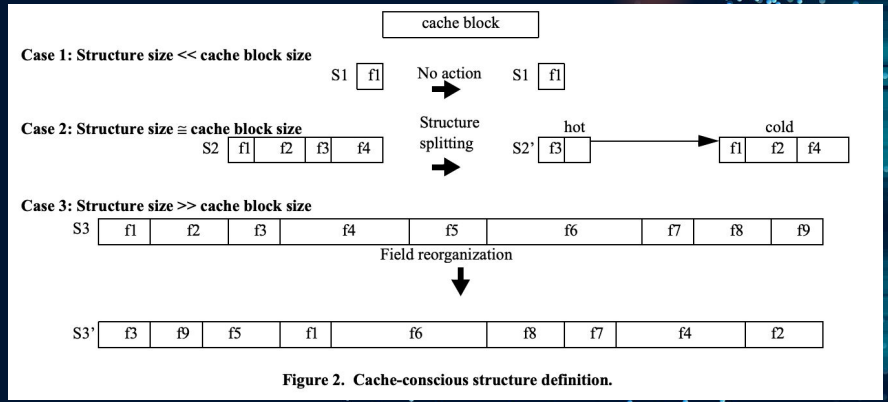


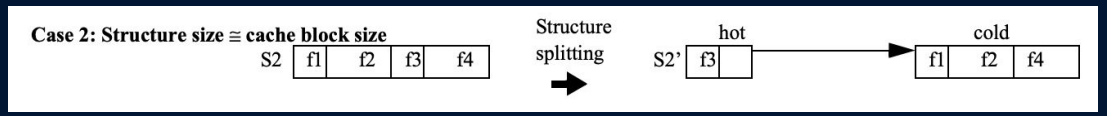
Figure 2. Cache-conscious structure definition.

Technique 1: Structure (Class) Splitting

- Separate fields into **hot** (frequently used) and **cold** (rarely used)
- Hot fields stay in the original class
- Cold fields move to a cold companion class
- Reduces cache pressure for medium-sized structures (≈1–2 cache blocks)

Structure Splitting - Hot vs Cold

- **Hot fields:** frequently accessed, performance-critical
- **Cold fields:** rarely accessed, error-handling or infrequently used metadata
- **Goal:** Pack hot fields tightly to fit inside 1 cache block
- Cold fields moved to cold-class → accessed via **1 extra** indirection



Structure Splitting - Heuristics & Decision Criteria

- Only split “live” & suitably sized classes
- Fields classified as hot vs cold by access frequency
- Avoid over-splitting using Temperature Differential (TD)
- Require sufficiently large cold portion (≥ 8 bytes)

$$A_i > LS / (100 * C)$$

A_i = total field accesses to class i
LS = total program field accesses
C = number of classes with ≥1 field access

$$TD(class_i) = max(hot(class_i)) - 2 * \Sigma cold(class_i) >> 0$$

Structure Splitting - Implementation

```
class A {
    protected long a1;
    public int a2;
    static int a3;
    public float a4;
    private int a5;
    A(){
        ...
        a4 = ..;
    }
    ...
}
class B extends A {
    public long b1;
    private short b2;
    public long b3;
    B(){
        b3 = a1 + 7;
        ...
    }
    ...
}
```

```
class A {
    public int a2;
    static int a3;
    public cld_A cld_A_ref;
    A(){
        cld_A_ref = new cld_A();
        ...
        cld_A_ref.a4 = ..;
    }
    ...
}
class B extends A {
    public long b3;
    public cld_B cld_B_ref;
    B(){
        cld_B_ref = new cld_B();
        b3 = cld_A_ref.a1 + 7;
        ...
    }
    ...
}
```

```
class cld_A {
    public long a1;
    public float a4;
    public int a5;
    cld_A(){...}
}

class cld_B {
    public long b1;
    public short b2;
    cld_B(){...}
}
```

Figure 5. Program transformation.

Real-world results: Java Benchmarks

- Java objects are mostly small, but many classes have hot/cold field profiles
- Splitting applied to 26–100% of candidate classes
- L2 cache miss rate reduced by 10–27%
- Execution time improved by 6–18% (beyond prior techniques)

Technique 2: Field Reordering for Large C Structures

- Targets large C structs that span multiple cache blocks
- Fields are often grouped by meaning, not by runtime access patterns
- Idea: reorder fields so temporally related fields share cache lines
- Implemented via an offline analysis tool called bbcache

bbcache: How It Works

- Build a structure access database from static analysis + profiling
- Construct a field affinity graph: nodes = fields, edges = temporal co-access frequency
- Greedy layout algorithm maximizes a “configuration locality” score
- Evaluate layouts via cache block pressure and cache utilization metrics

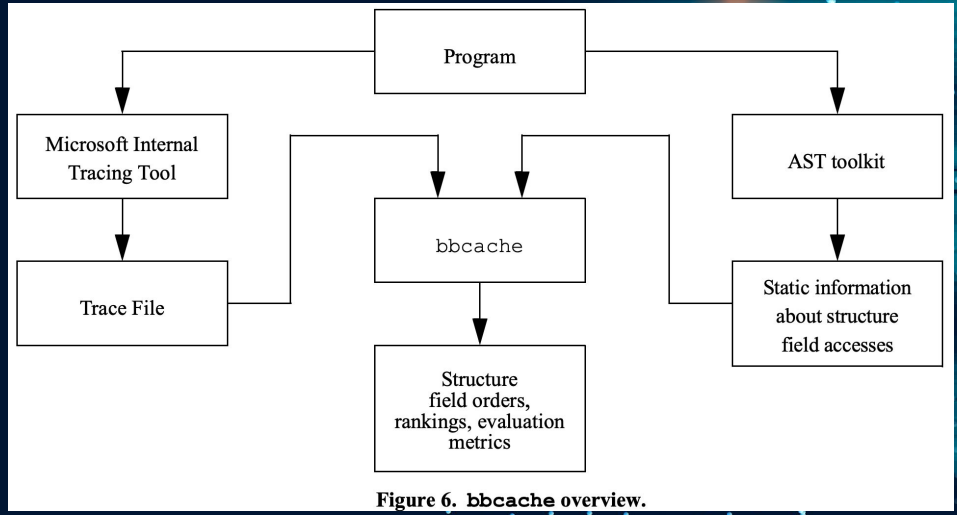


Figure 6. bbcache overview.

Field Reordering Metrics

- Need a way to compare field layouts without re-running the whole app
- Partition execution into short time windows (e.g., 100 ms)
- Two key metrics:
  - cache block pressure
    - average number of distinct blocks touched per window
  - cache block utilization
    - Utilization: average fraction of each touched block's bytes that are actually used
    - Good layouts → lower pressure, higher utilization

SQL Server Case Study

- Microsoft SQL Server 7.0 running TPC-C
- ~2000 structs; 163 account for >98% of structure accesses
- Many structs constrained (on-disk formats, casting); only a few are safe to reorder
- Reordered **5 key structs**; improved utilization/pressure metrics and **gained 2–3% throughput**

Structure	Cache block utilization (original order)	Cache block utilization (recommended order)	Cache pressure (original order)	Cache pressure (recommended order)
ExecCxt	0.607	0.711	4.216	3.173
SargMgr	0.714	0.992	1.753	0.876
Pss	0.589	0.643	8.611	5.312
Xdes	0.615	0.738	2.734	1.553
Buf	0.698	0.730	2.165	1.670

Limitations and Practical Considerations of CCSD

- Both techniques **heavily** rely on profiling
- Low-level constraints:
  - ABIs, on-disk formats, and type punning limit what can be changed
- Overheads:
  - more objects and pointers (for splitting), analysis and build-time cost
- Optimizations are hardware- and cache-configuration-specific

Takeaways and Lessons

- Data layout matters as much as algorithms
- Two complementary techniques:
  - Structure Splitting for medium-sized objects
  - Field Reordering for large objects
- Both built on temporal locality profiling
- Significant performance wins:
  - 10–27% fewer L2 misses, 6–18% faster Java
  - 2–3% SQL Server speedup
- Simple heuristics → robust, input-stable, and low-risk
- Layout optimizations integrate well with existing systems