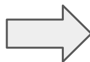
Com-CAS: Effective Cache Apportioning Under Compiler Guidance

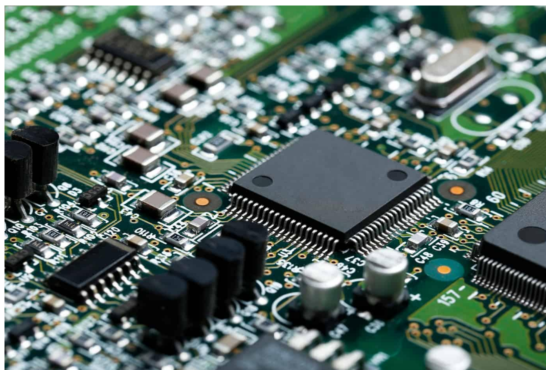
Bodhisatwa Chatterjee   Sharjeel Khan   Santosh Pande

Georgia Tech

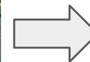
Shared Cache = Inter-Application Interference

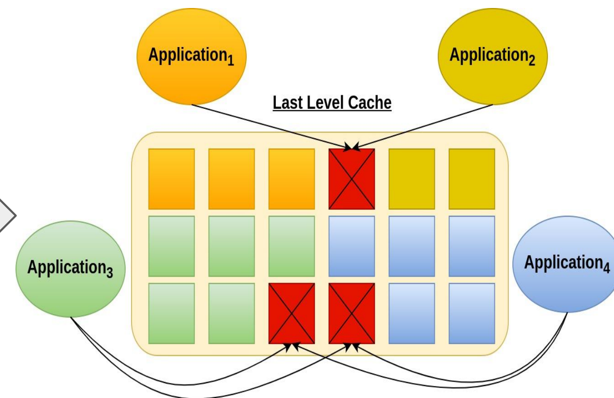
HPC Servers/Data Centers





Shared Last-Level Cache (LLC)



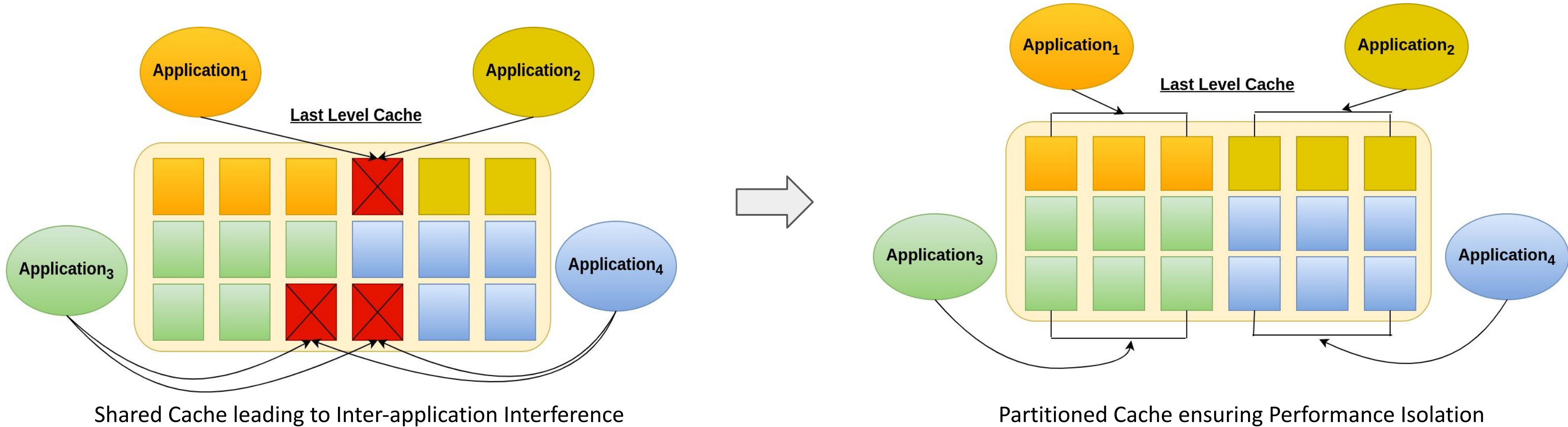


Inter-Application Interference

⏩ HPC Servers facilitate concurrent execution by sharing resources (Caches, Memory Bandwidth, Inter-connect) among applications.

⏩ Sharing the Last-Level Cache (LLC) results in **Inter-Application Interference**, where multiple applications map to same cache line, resulting in conflict misses.

Cache Partitioning = Performance Isolation



Shared Cache leading to Inter-application Interference

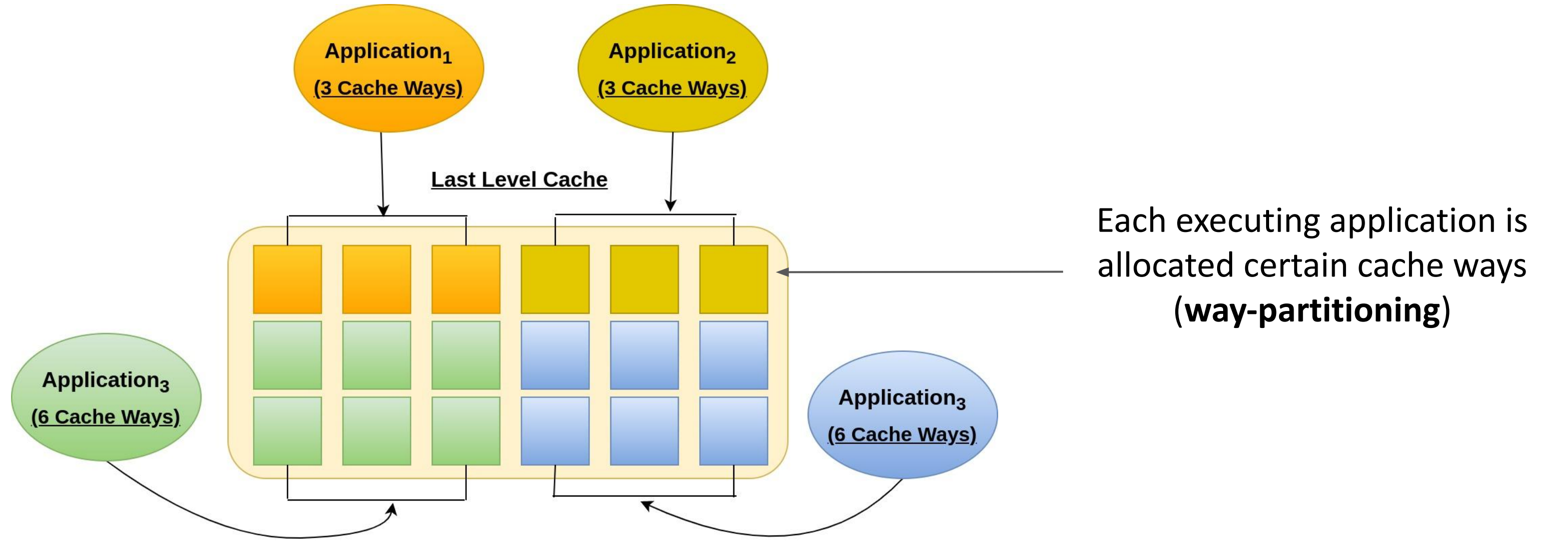
Partitioned Cache ensuring Performance Isolation

⏩ **Cache Partitioning** divides the LLC among the co-executing applications in the system.

⏩ This secures dedicated regions of cache memory to high-priority cache-intensive applications, resulting in superior application performance and enhanced system throughput.

⏩ It can also be leveraged to boost system utilization, improve power & energy consumption, fair resource allocation, worst-case timing analysis, etc

How should the Last-Level Cache be partitioned?

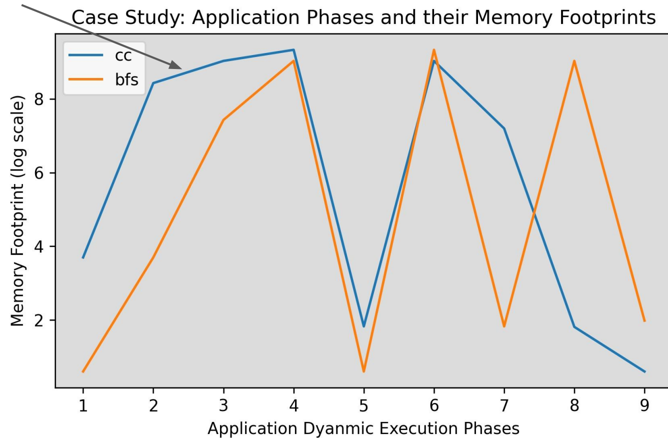


Each executing application is allocated certain cache ways (**way-partitioning**)

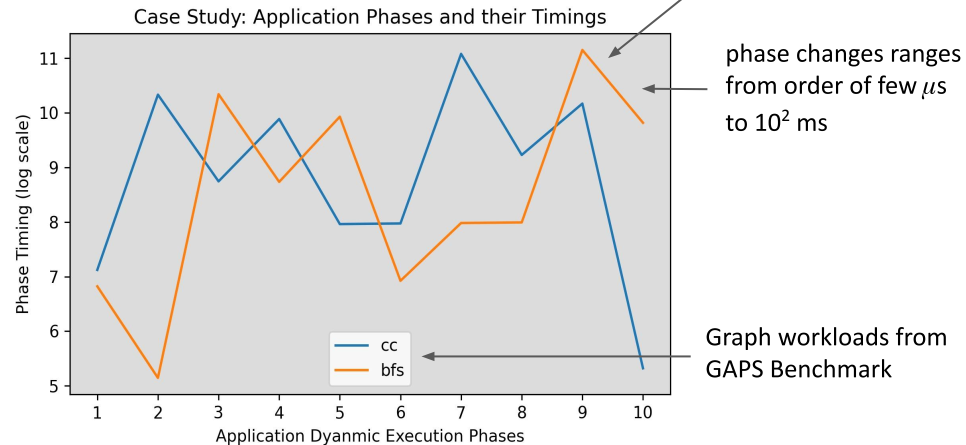
- The amount of cache allocated to each application should satisfy individual cache requirements
- **How do we determine an application's requirements throughout its execution?**

Modern Workloads = Dynamic Phase Behavior

Rapid memory footprint transitioning in footprint log scale



Total execution phases and their timings are input-dependent

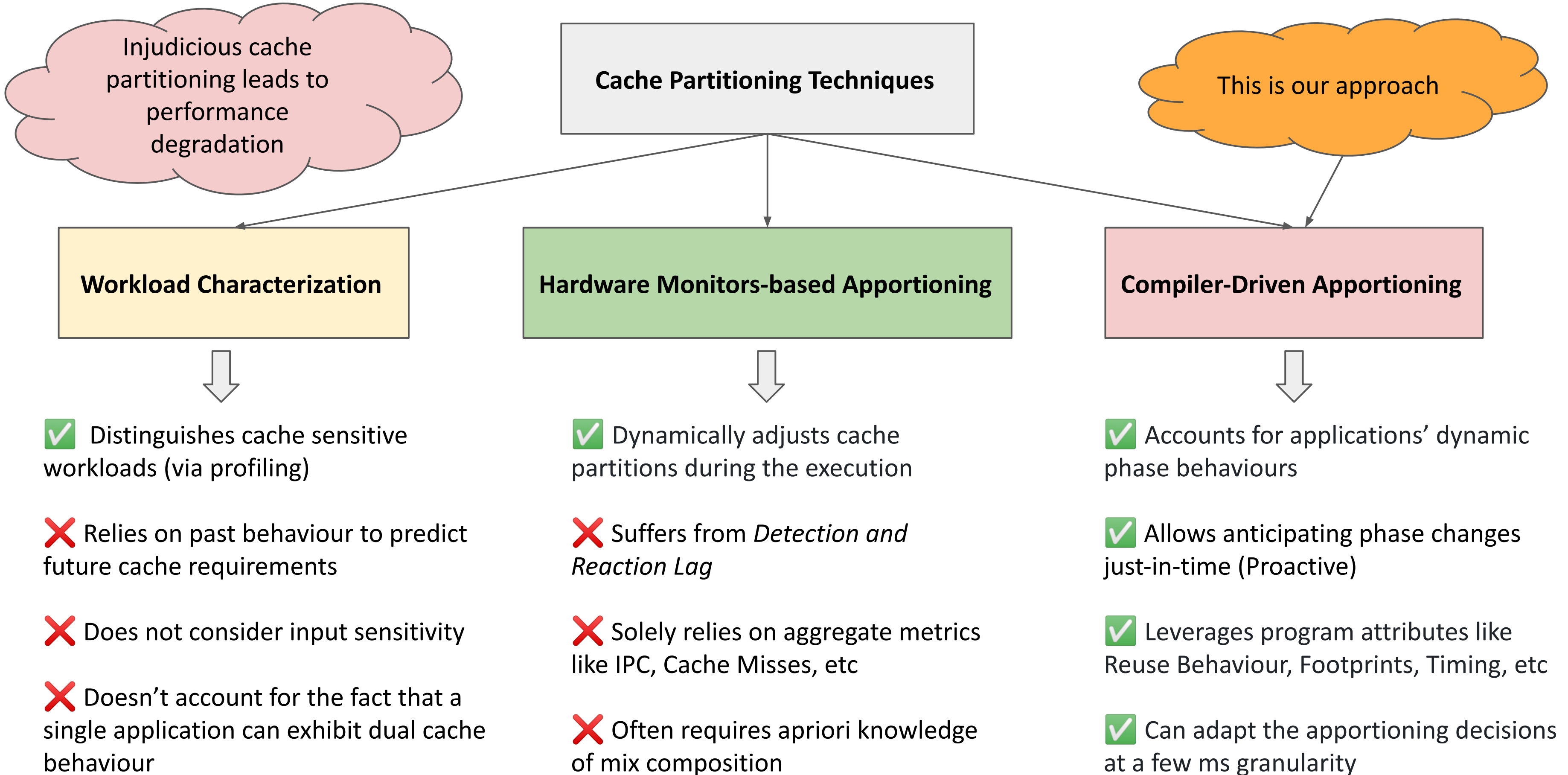


⏩ Modern workloads exhibit '**dynamic phase behaviour'** due to which their cache requirements rapidly changes throughout their execution cycle

⏩ These behaviours result from input dependencies, complex control flows, diverse memory referencing behaviours present in the application

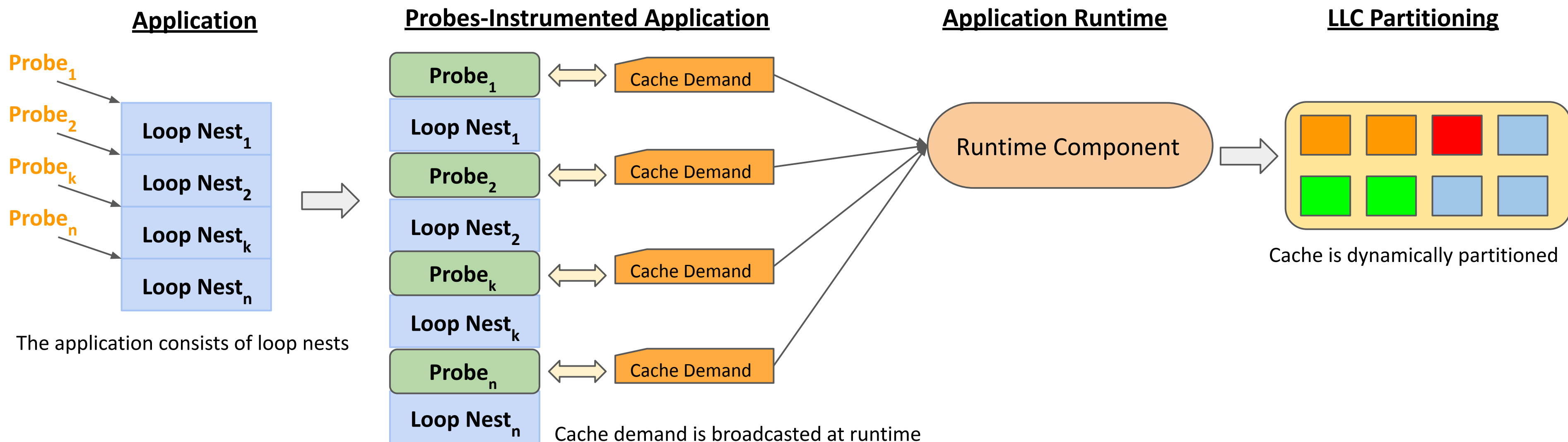
⏩ Even a single program region (such as a loop) can exhibit different behaviours upon different invocations.

Cache Partitioning Systems = Oblivious to Phase Behaviour



Compiler-Guided Cache Apportioning System (Com-CAS)

⏩ Applications' cache requirements are estimated just-in-time, through a combination of static & dynamic program attributes, by leveraging compiler analysis & machine learning.

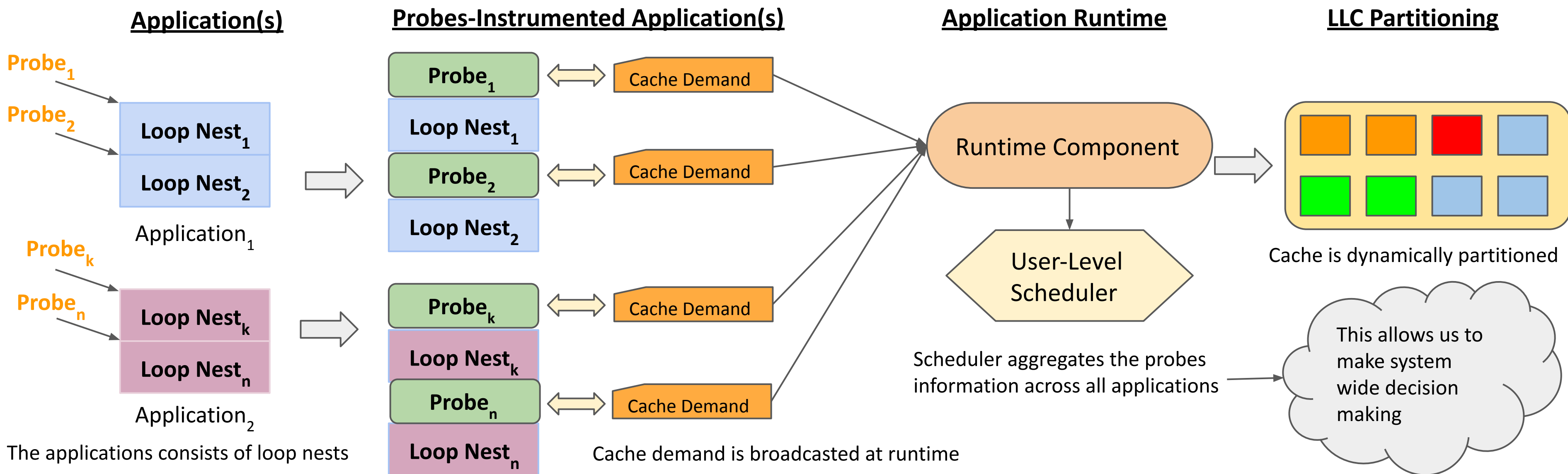


⏩ **Probes** are specialized library markers, which encapsulate the cache requirements, and are statically instrumented in the application (Static Generation).

⏩ These probes broadcast the cache requirements during runtime, which helps scheduler to make partitioning decisions (Dynamic Instantiation).

Compiler-Guided Cache Apportioning System (Com-CAS)

⏩ Applications' cache requirements are estimated just-in-time, through a combination of static & dynamic program attributes, by leveraging compiler analysis & machine learning.

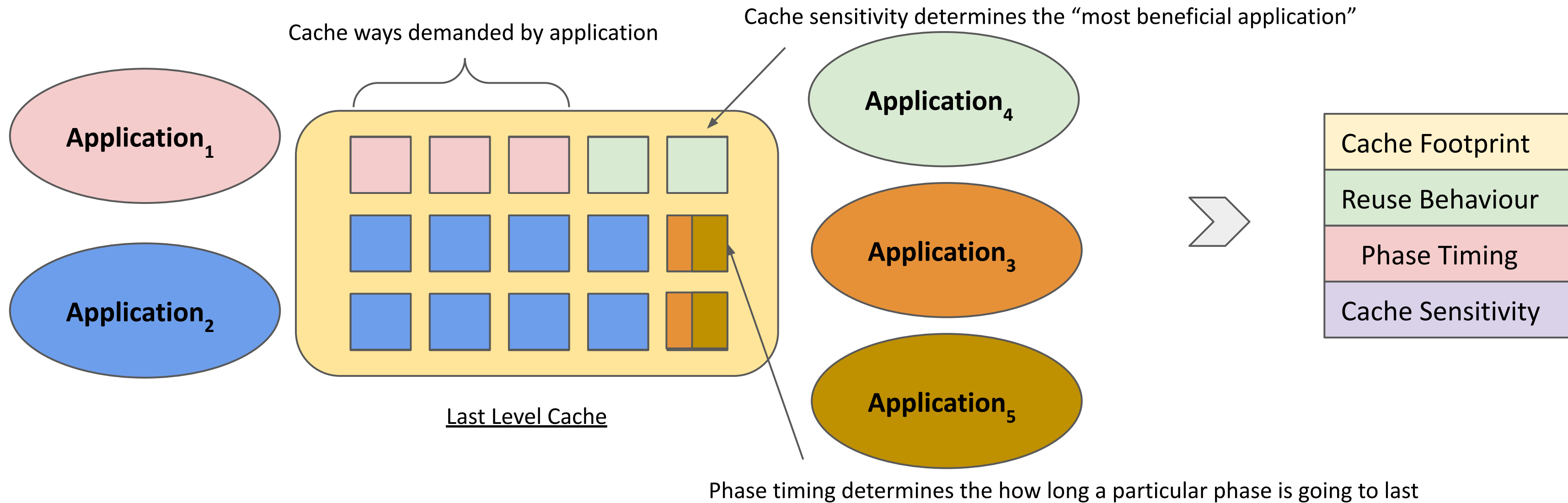


⏩ **Probes** are specialized library markers, which encapsulate the cache requirements, and are statically instrumented in the application (Static Generation).

⏩ These probes broadcast the cache requirements during runtime, which helps scheduler to make partitioning decisions (Dynamic Instantiation).

Quantifying Cache Requirements

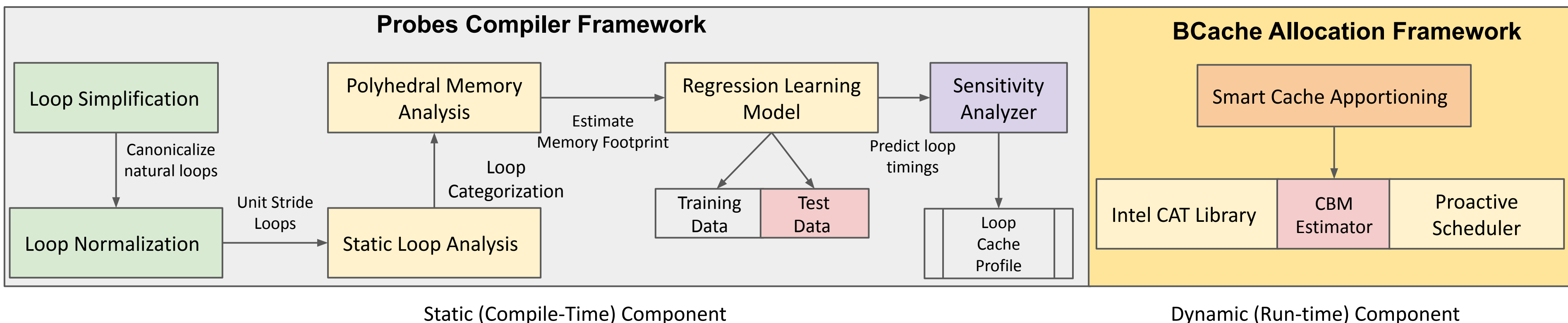
⏩ Probes estimates program attributes that dictate an application's cache requirements.



⏩ These four program attributes (cache footprint, reuse behaviour, phase timing, cache sensitivity) are expressed as closed-form expressions, and are computed via compiler analysis coupled with learning algorithms

Com-CAS : Smart and Proactive Cache Partitioning

⏩ Compile-time and Runtime Cooperative Cache Partitioning System. It consists of front-end and back-end.



⏩ **Probes Compiler Framework** represents Com-CAS's frontend. It is responsible for estimating the program attributes and encapsulating them, and instrumenting the probes in the application.

⏩ **BCache Allocation Framework** represents Com-CAS's backend. It consists of phase-aware cache allocation algorithms, and a proactive workload scheduler that aggregate probes information and determines LLC partitions.

⏩ **Intel CAT** is leveraged to perform the actual cache partitions - the scheduler interacts with it via customized library.

Probes Compiler Framework : Predicting Phase Timing

⏩ **Loop-Timing** is defined as the time taken for executing an entire loop-nest. It helps us to determine how long a particular phase will last.

⏩ **Theorem.** *For a normalized loop nest L with n inner-nested loops with individual upper-bounds* $\{U_1, U_2, .., U_n\}$ *the timing T_c is given by the linear equation:*

$$T_c = U^T C = c_0 + c_1 u_1 + c_2 u_2 + \ldots + c_n u_n$$

where $C = \{c_0, c_1, ..c_n\}$ are *learnable parameters*

$U = \{u_0, u_1, .., u_n\}$ is the *feature vector*

$u_i = \prod_{k=1}^{i} U_k$ represents the *individual feature*

⏩ During runtime, the actual loop-bounds are plugged into this equation to generate the phase-time.

Probes Compiler Framework : Estimating Memory Footprint

⏩ **Memory footprint** determines the amount of cache that will be utilized by a loop-nest during an execution phase.

⏩ Polyhedral analysis generates memory footprint equations of the form:

$$[X] \rightarrow \{m(X) : 0 < X < N\}$$

m(X) represents the dynamic memory accesses
N is the expected iterations of the loop nest

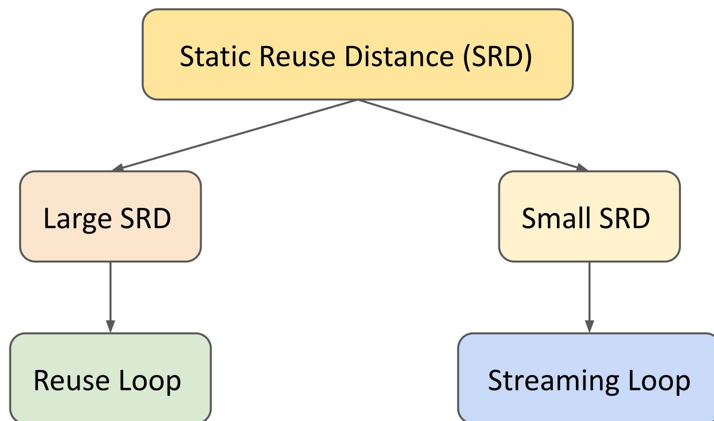
⏩ For non-affine loops, the memory footprints and timings are taken as an average on the training input sets.

⏩ We found that this works quite well in practice, as Com-CAS works on *aggregate cache requirements of all the co-executing processes* in the system. Thus, approximate values for footprints and timings are sufficient for determining system-wide cache-apportions.

Probes Compiler Framework : Classifying Data Reuse

⏩ To determine the amount of cache required by a loop-nest for maximizing locality, we need to obtain a sense of reuse behaviour exhibited by the loop-nest.

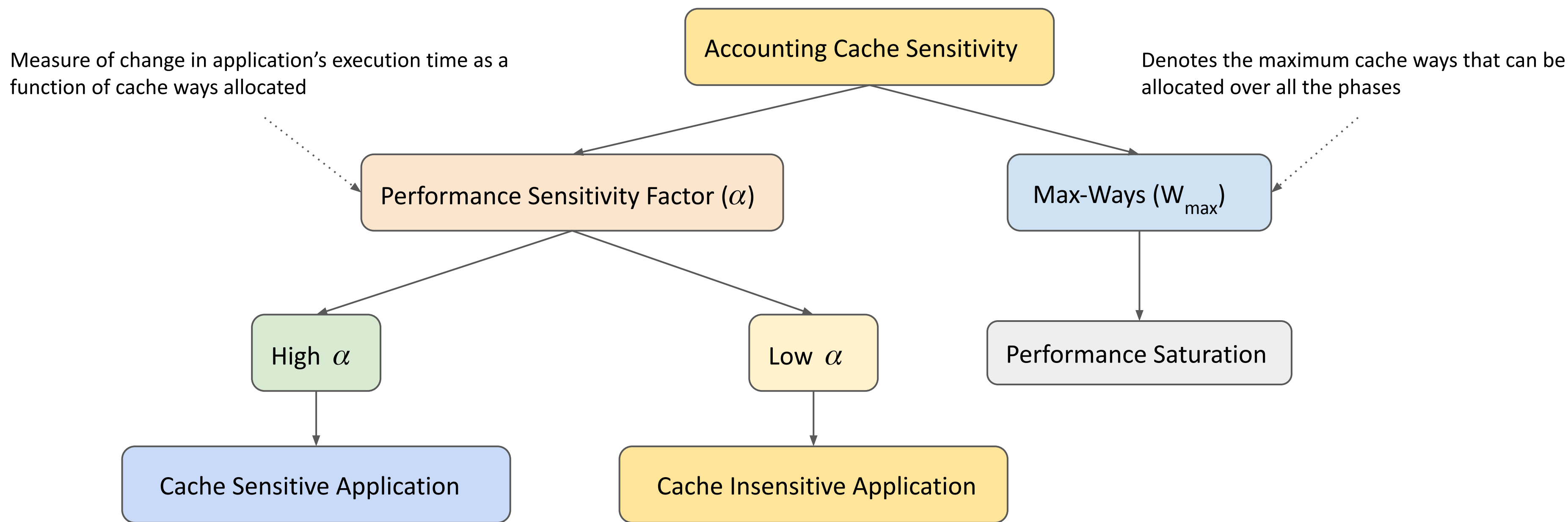
⏩ To classify reuse behaviour, Probes Framework uses **Static Reuse Distance (SRD)**.



⏩ For large reuse distances, large cache resources must be allocated so that the reused data will be found in the cache. For small reuse distances, data which is reused only after a couple of iterations will be found in the cache.

Probes Compiler Framework : Accounting Cache Sensitivity

⏩ Com-CAS accounts for cache-sensitivity by defining **performance sensitivity factor** (α) and **max-ways** for entire application.

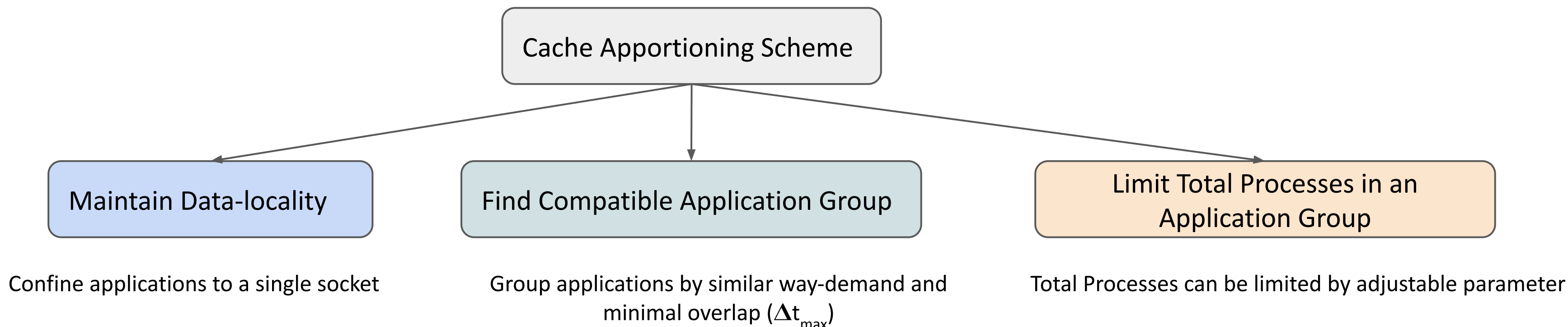


⏩ These metrics are application-wide, and are simply meant to guide the apportioning decisions.

BCache Allocation Framework : Cache Apportioning Scheme

⏩ Goal is to obtain efficient cache partitions for diverse application mixes, based on their execution phases.

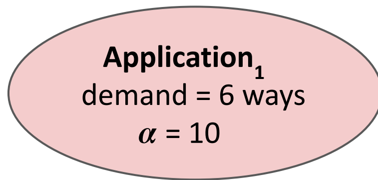
⏩ An application executing a reuse loop with a higher value of memory footprint should be allocated a greater portion (isolated) of the LLC, compared to another application executing a streaming loop.

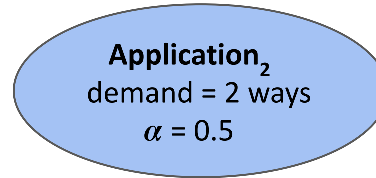


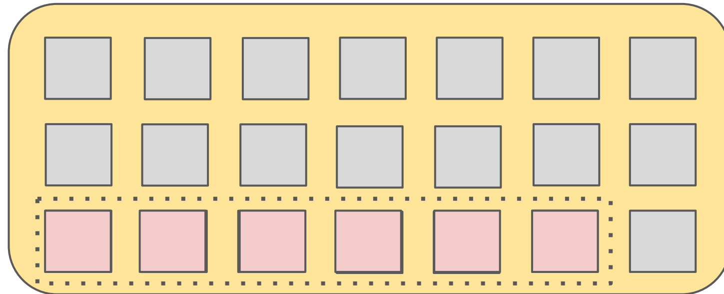
⏩ BCache Framework uses an unit-based fractional cache apportioning scheme - each application will be allocated a fraction of the LLC, which is measured by estimating how much they contribute to the entire memory footprint.

BCache Allocation Framework : Phase-Aware Cache Allocation

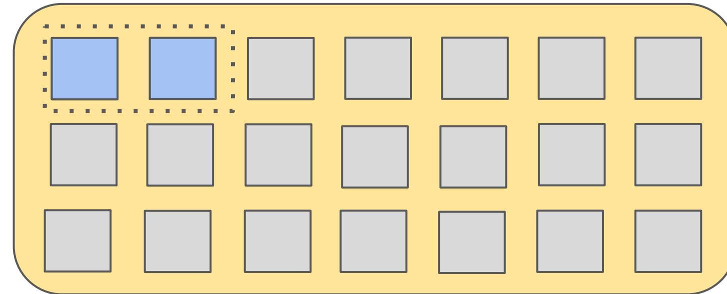
- Applications are first grouped into different sockets based on their α-values.
- A compatible CLOS (group) is selected based on nature of loop (*reuse - minimal overlap/ stream - grouping*)







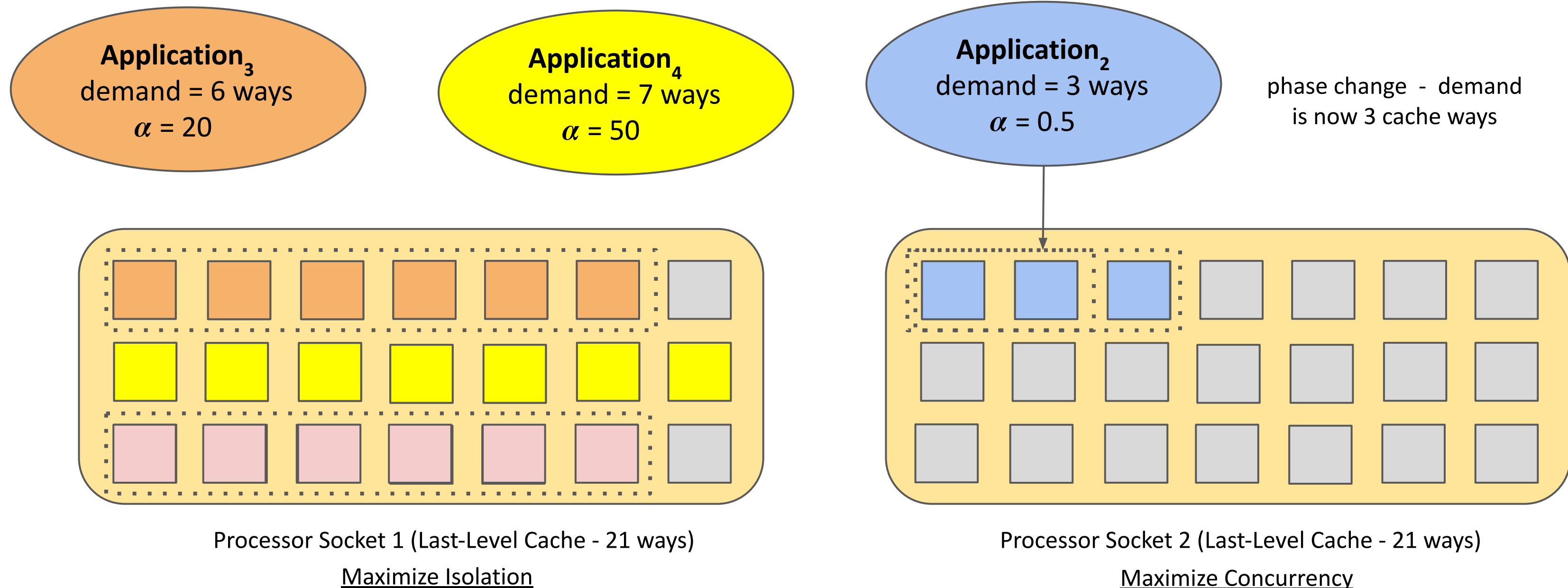
Processor Socket 1 (Last-Level Cache - 21 ways)
Maximize Isolation



Processor Socket 2 (Last-Level Cache - 21 ways)
Maximize Concurrency

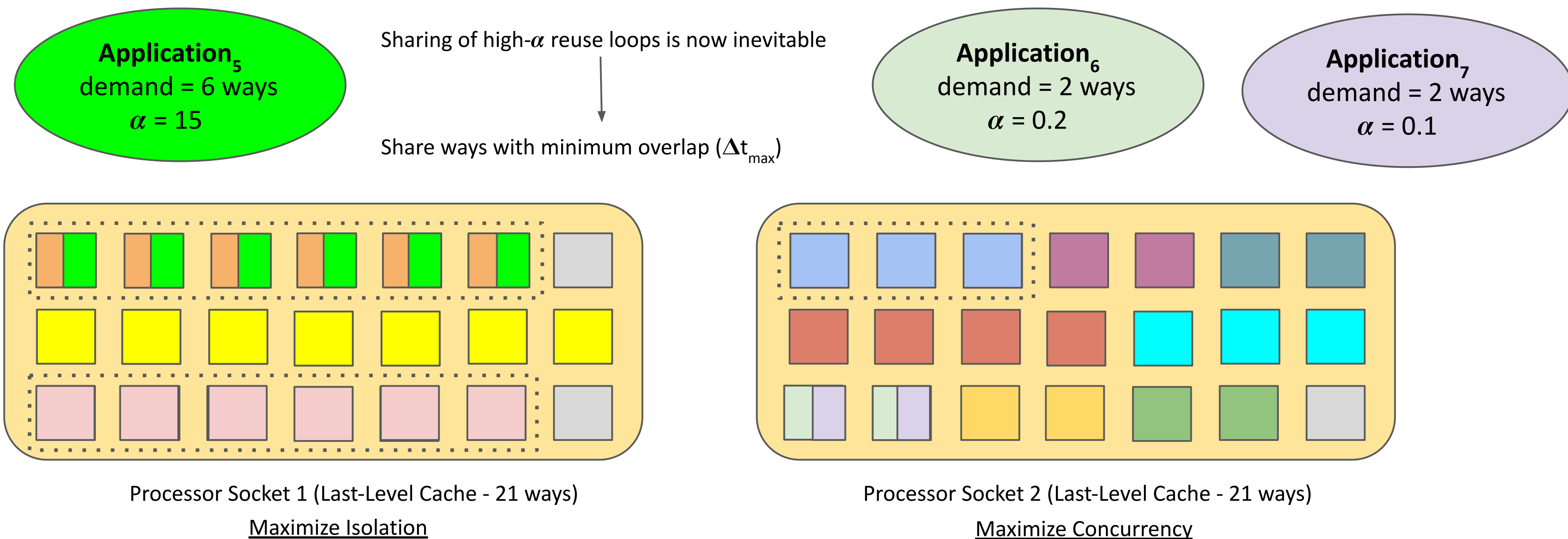
BCache Allocation Framework : Phase-Aware Cache Allocation

- Applications are first grouped into different sockets based on their α-values.
- A compatible CLOS (group) is selected based on nature of loop (*reuse - minimal overlap/ stream - grouping*)



BCache Allocation Framework : Phase-Aware Cache Allocation

- Applications are first grouped into different sockets based on their α-values.
- A compatible CLOS (group) is selected based on nature of loop (*reuse - minimal overlap/ stream - grouping*)



Com-CAS: Experimental Evaluation

- Com-CAS was evaluated with 45 application mixes from *GABPS*[1], *Polybench*[2], *Rodinia*[3], and *SPEC 2017*[4]
- **Four different baselines** were used to evaluate Com-CAS's performance:

Partition Scheme	Policy Type	Interval	Throughput Improvement (Average)
Unpartitioned Cache	NA	NA	15% speedup
Max-Ways Partitioning	Static	NA	21% speedup
HW Performance Counter	Dynamic	250 ms	32% speedup
Kpart[5]	Dynamic	20^6 cycles	20% speedup

Com-CAS's performance improvement compared to each baseline

- Experiments on Dell PowerEdge R440 server with Intel Xeon Gold 5117 processors with Intel CAT, 28 cores, 11-way set-associative, 19 MB shared LLC, running Ubuntu 18.04

[1] http://gap.cs.berkeley.edu/benchmark.html

[2] https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

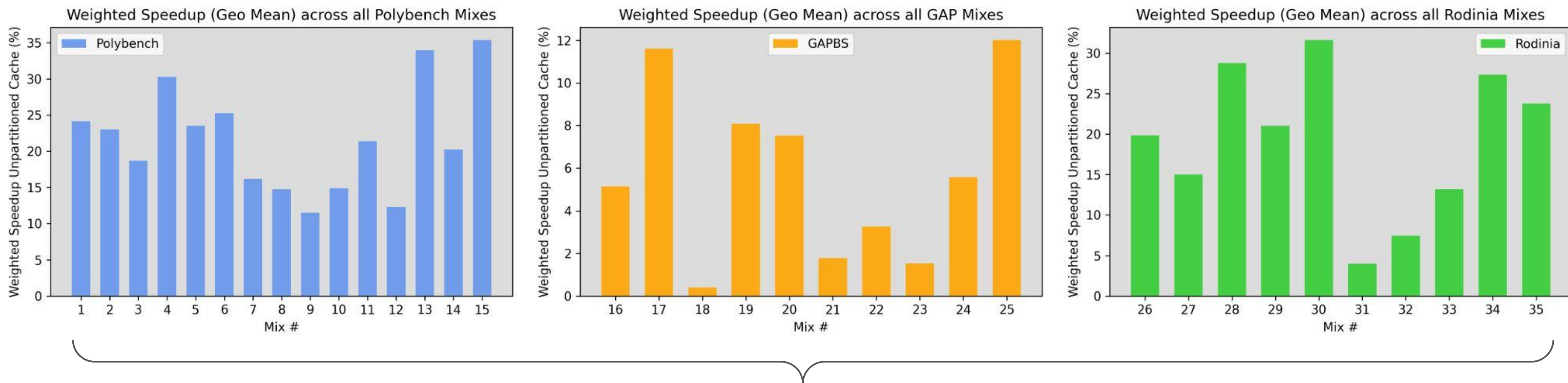
[3] https://www.cs.virginia.edu/rodinia/doku.php

[4] https://www.spec.org/cpu2017/

[5] https://www.csail.mit.edu/research/kpart-novel-technique-partitioning-shared-caches

Com-CAS: Throughput Enhancement for Unpartitioned Cache

⏩ The largest performance gains are in heavy-mixes, where the workload resource requirement saturates the system and judiciously partitioning the cache is highly contingent upon utilizing dynamic phase attributes.

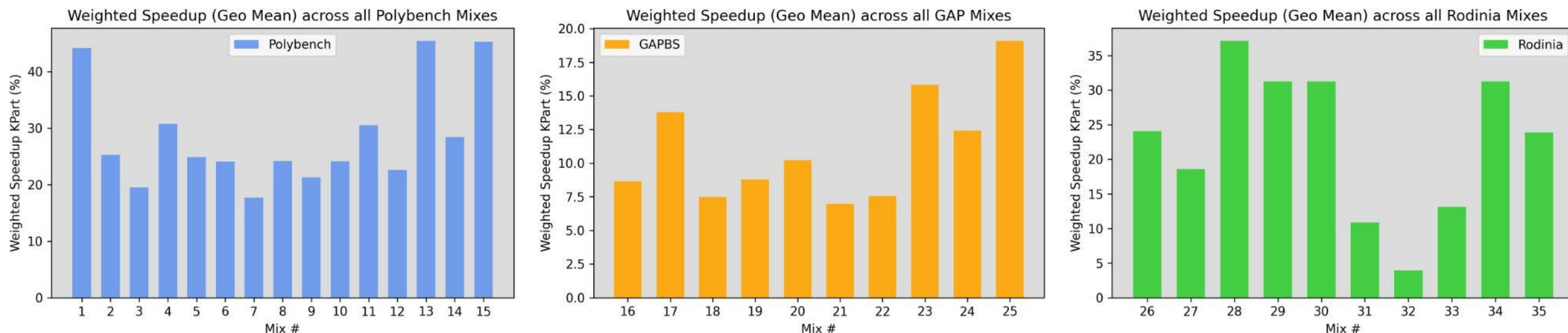


The average performance improvement over all the mixes were 15%

⏩ Com-CAS achieves effective cache allocation and superior scheduling particularly when the workload resource demands are saturated and it prevents overwhelming of the system.

Com-CAS: Throughput Enhancement versus KPart

⏩ Kpart[5] groups applications in distinctive clusters by checking the reduction in combined cache miss among applications by profiling. It periodically updates them periodically per 20^6 cycles of instructions.



The average performance improvement over all the mixes were 20%. Performance gap arises due to KPart's non-adaptability of to the varied phase timings and footprints

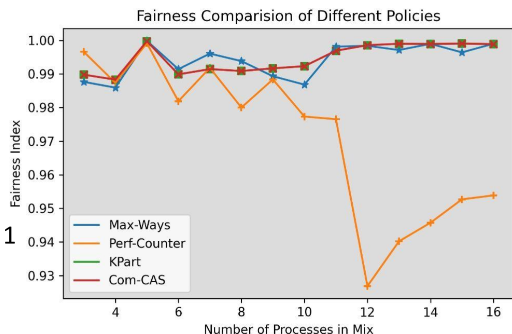
⏩ KPart essentially treats each application like a 'black-box' and attempts to find 'best cluster fit' for a certain application mix, while Com-CAS is assisted by a compiler framework that analyzes each application at loop nest granularity.

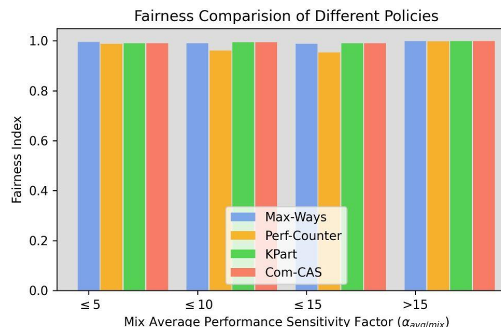
[5] https://www.csail.mit.edu/research/kpart-novel-technique-partitioning-shared-caches

Com-CAS: Cache-Misses, Fairness and Individual Latencies

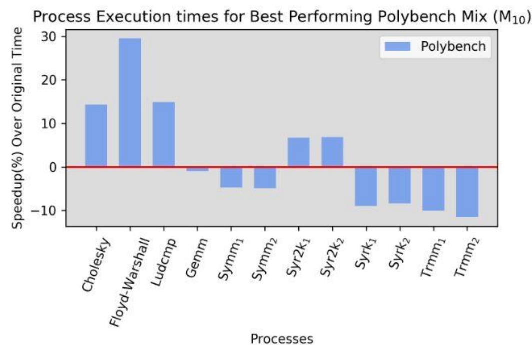
⏩ For Cache Misses, the general trend is that the reduction in LLC cache misses are shifted towards reuse-based applications that "need a greater amount of cache".

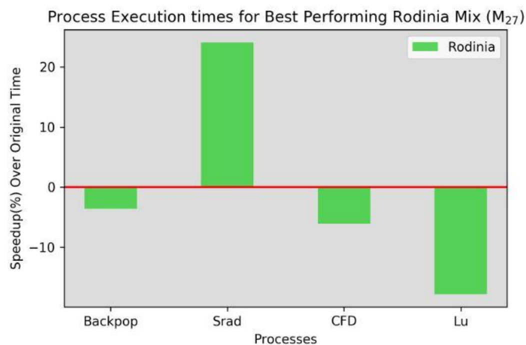
Com-CAS's Fairness Index is close to 1

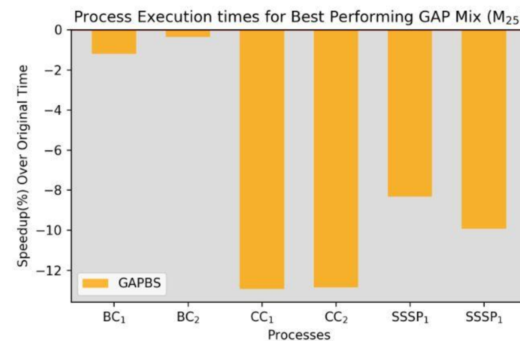




All processes show minimum execution latency degradation







Individual process latencies with original-unmixed time in representative mixes from each benchmark (#10, #25, #27).

Com-CAS: Conclusion

⏩ Com-CAS: **Com**piler-Guided **C**ache **A**pportioning **S**ystem

- ❑ Effective apportioning of the shared LLC leveraging Intel CAT
- ❑ **Probes Compiler Framework** evaluates cache-attributes such as reuse behaviour, cache footprints, loop timings and cache sensitivity and relays them.
- ❑ **BCache Allocation Framework** uses allocation algorithms to dynamically partitions the cache and schedules processes

⏩ Com-CAS improved average throughput by 15% on unpartitioned cache system, and 20% on state-of-art KPart

⏩ With improved throughput, minimal latency degradation, and reduced process interference, we contend that the proposed Com-CAS is a viable system for multi-tenant setting

BACKUP SLIDES : Intel CAT

⏩ Intel **C**ache **A**llocation **T**echnology (CAT) is a part of Intel Resource Director Technology (RDT)

⏩ Goal is to provide extended control/visibility over shared resources to users

⏩ Intel CAT is a reconfigurable implementation of hardware way partitioning - it lets the user to specify custom cache partitioning configurations to different applications

⏩ Cache usage of any application can be adjusted via **CLOS** and **Capacity Bitmasks (CMBs)**

- ❑ **Class of Service (CLOS)**: Applications in the same CLOS share the same partition
- ❑ **Capacity Bitmasks (CBM)**: The exact number of ways is specified through n-dimensional bitvector

⏩Additional Hardware Components and some minor changes are done in Linux Kernel to support Intel CAT

BACKUP SLIDES : Com-CAS's Extension to Other Hardwares

⏩ Com-CAS relies on Intel CAT to perform the actual cache partitioning.

⏩ The Probes Compiler Framework (& its compiler analysis), BCache Allocation Framework (& its apportioning scheme and algorithms) are independent of the underlying architecture.

⏩The library interface that interacts between the proactive scheduler and Intel CAT is architecture-dependent, and is the only component that requires re-engineering for other architectures.

⏩Some ARM architecture also supports reconfigurable cache partitioning[6].

[6] https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/l3-cache-partitioning

BACKUP SLIDES: Com-CAS Overheads

⏩ The Com-CAS Framework has the following sources of runtime overheads:

- ❑ Short probe-library calls during information broadcast that are in range of 10 µs (less than 1% overheads)
- ❑ Training the regression models that adds ~ 120 secs, and embedding adds ~ 250 secs to compilation time.

⏩ Linear models were chosen to minimize overheads.

⏩ Probe calls are hoisted to outer-most loop's pre-header to avoid the scheduler getting overwhelmed by excessive calls.