Locating Cache Performance Bottlenecks Using Data Profiling

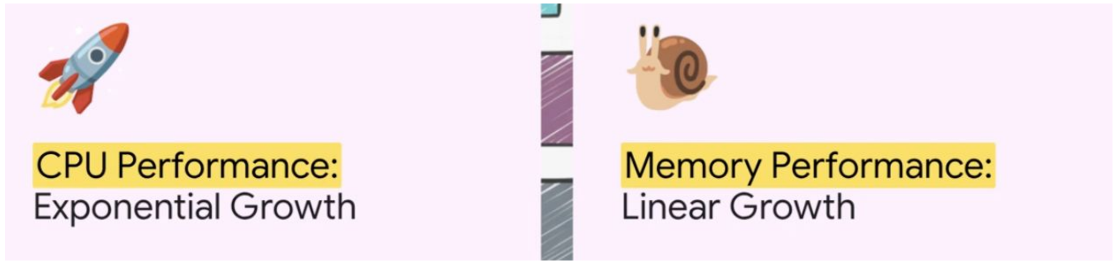
Group 4: Garv Shah, Sydney Shanahan, Hillary Luan, Blake Potvin, Prakhar Gupta

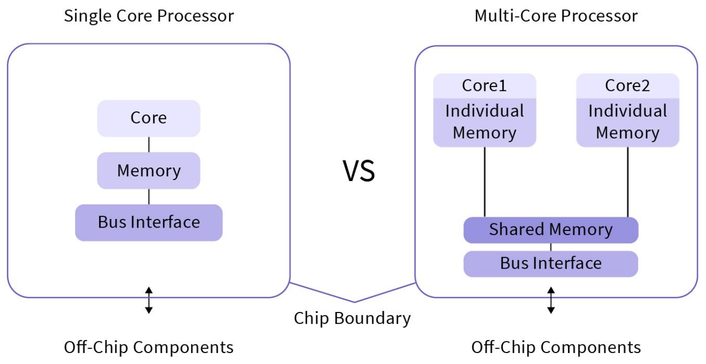
Introduction & Motivation

Multicore system: processor with multiple cores (instruction executors) that can read/execute program instructions simultaneously

Performance on Multicore Systems

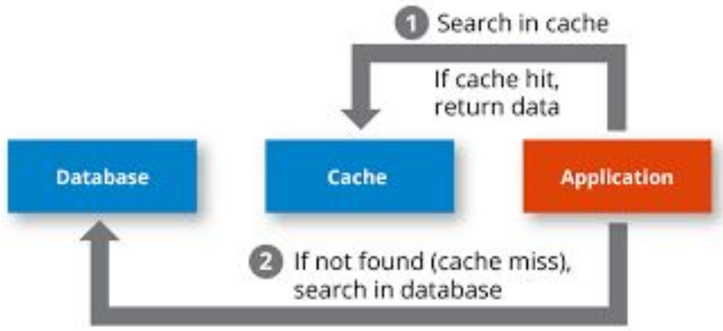
- Modern CPUs frequently stall waiting for memory
- Cache misses dominate performance cost
- Multicore systems worsen the stalling issue



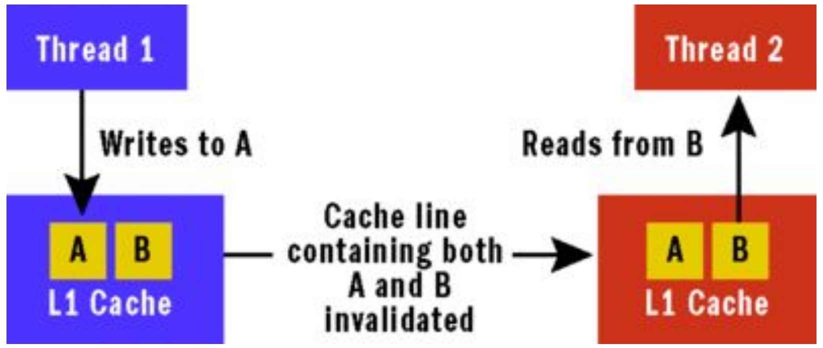


Cache Miss Types (Glossary)

- Misses (Uniprocessor - One Core)
  - Compulsory misses — "cold start", cache is empty so first reference to an address is going to miss
  - Conflict misses — data we want got evicted earlier from cache
  - Capacity misses — cache is too small to hold all the data (wouldn't have missed if cache were bigger)
- Misses (Multiple Cores)
  - True sharing misses
    - multiple cores read/write same memory location
  - False sharing misses
    - multiple cores read/write different memory location **but** on same cache line

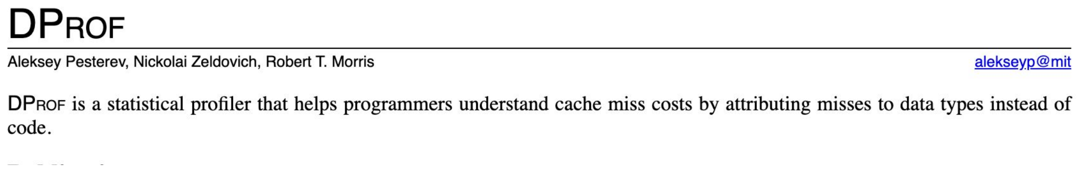


1 Search in cache
If cache hit, return data
Database
Cache
Application
2 If not found (cache miss), search in database



Why This Matters

- Typical Cache Profilers: Code Location
- What about instructions spread over locations in application accessing/missing the same exact data type?
- Solution:



DPROF

Aleksey Pesterev, Nickolai Zeldovich, Robert T. Morris alekseyp@mit

DPROF is a statistical profiler that helps programmers understand cache miss costs by attributing misses to data types instead of code.

- DProf Provides:
  - Miss cause classification (true/false sharing, conflict, capacity) -> important as mitigating a miss depends on its type
  - User-friendly interface
  - associates cache misses with **data types** not location

How DProf Works - High Level Overview

Implemented as a set of Linux kernel modifications & modules for data collection
User-space program for programmer to analyze results

Collect Memory Access Samples

Uses AMD's Instruction Based Sampling (IBS) hardware feature

Resolve Addresses to Data Types

Resolves memory address to C data type for counting and profiling

Collect Object Access Histories

Traces instructions that reference object

Path Trace Generation

Combine previous two memory datasets into path traces for each data type

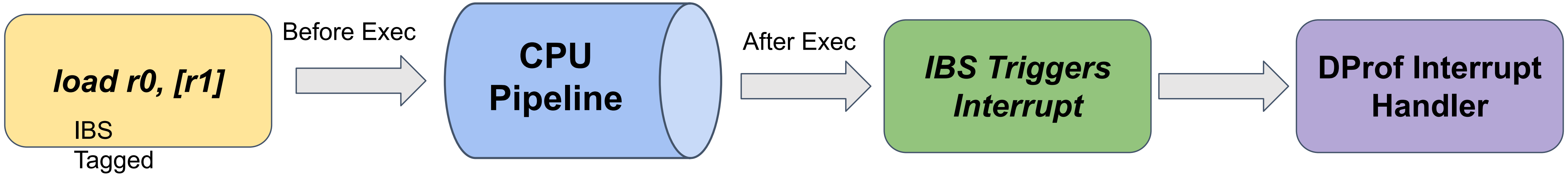
Generate DProf Views

Present data in a viewable and understandable format for the user

Collecting Memory Address Samples

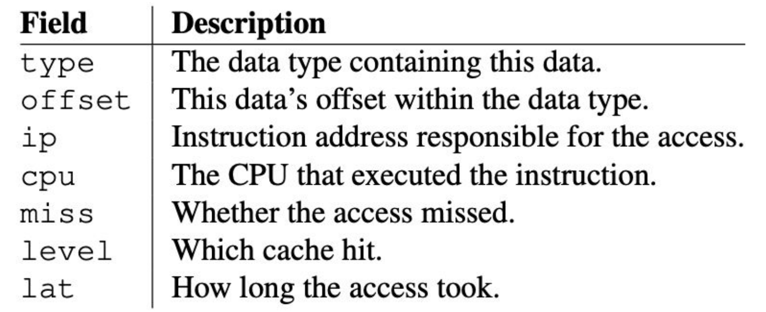
Captures the **Cost** of a memory access & Hot Data Types

AMD IBS randomly “tags” a micro-op for sampling before entering CPU pipeline.



Tagged Micro-ops Track The Following during Execution :

- Cache Misses
- Branch Mispredictions
- Memory Access Latency



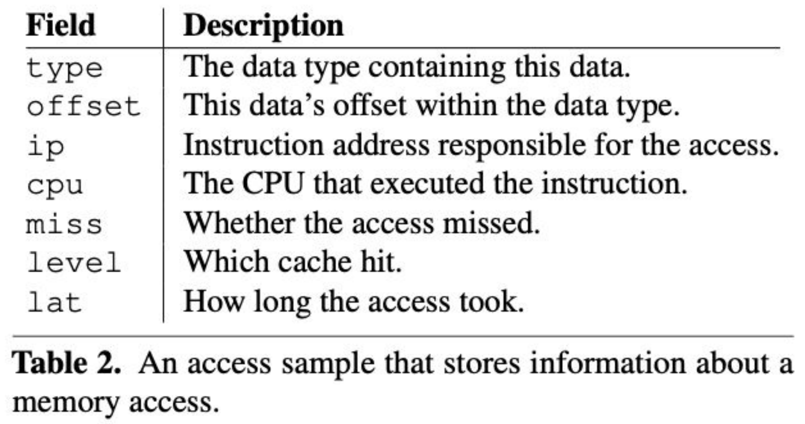
Field	Description
`type`	The data type containing this data.
`offset`	This data’s offset within the data type.
`ip`	Instruction address responsible for the access.
`cpu`	The CPU that executed the instruction.
`miss`	Whether the access missed.
`level`	Which cache hit.
`lat`	How long the access took.

Table 2. An access sample that stores information about a memory access.

Resolve Addresses to Data Types

Goals:

- What object does this memory address refer to?
- What is this object's type?
- What field was accessed (e.g. C-style struct)



Field	Description
`type`	The data type containing this data.
`offset`	This data's offset within the data type.
`ip`	Instruction address responsible for the access.
`cpu`	The CPU that executed the instruction.
`miss`	Whether the access missed.
`level`	Which cache hit.
`lat`	How long the access took.

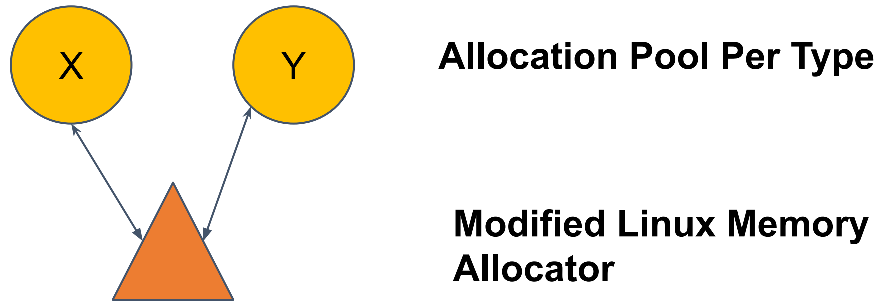
Table 2. An access sample that stores information about a memory access.

Statically Allocated Memory
(.data/.bss) w/ Debug Symbols

Check if IBS address within address ranges

<u>Simplified View of Debug Symbols</u>
0x34000 x1 struct X
0x34020 x2 struct X

Dynamically Allocated Memory



Offset = IBS Address - Base Address

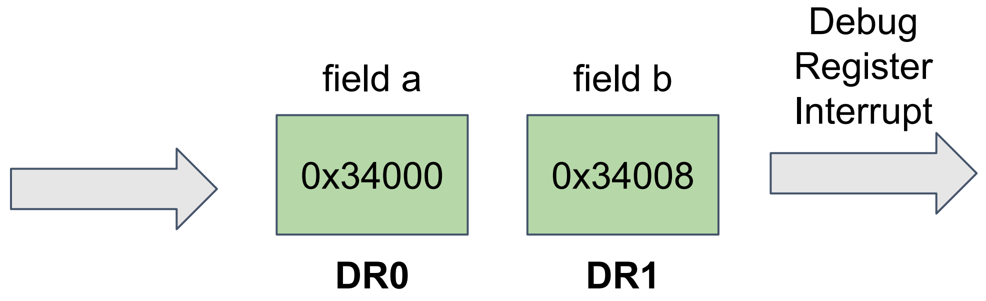
Object Access Histories

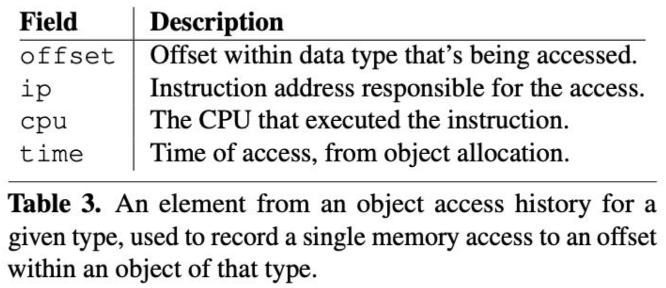
DProf utilizes **debug registers** to track object access by interrupting on every load/store instruction.

Problem: Only a limited number of debug registers exist.
DProf's Solution: Track only one object at a time and their respective fields.

Access Samples

```
0x34000  x1  struct X
0x34020  x2  struct X
0x34040  y1  struct Y
```





Field	Description
`offset`	Offset within data type that's being accessed.
`ip`	Instruction address responsible for the access.
`cpu`	The CPU that executed the instruction.
`time`	Time of access, from object allocation.

Table 3. An element from an object access history for a given type, used to record a single memory access to an offset within an object of that type.

```
struct X {
  long a;
  long b;
  int c;
}
```

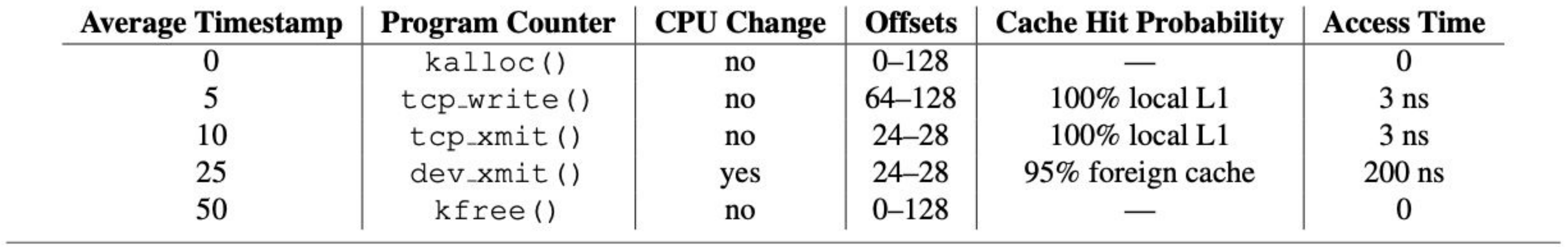
Path Trace Generation

1. Group up **Access Samples** with the same **type**, **offset**, and **instruction pointer**.

2. Augment **Object Access Histories** to include **miss**, **level**, and **lat** from access samples to records with the same **offset**, **type,** and **IP**.

3. Combine all augmented access histories with the same execution paths.

4. All of a the combined histories become a data type path trace.



Average Timestamp	Program Counter	CPU Change	Offsets	Cache Hit Probability	Access Time
0	`kalloc()`	no	0–128	—	0
5	`tcp_write()`	no	64–128	100% local L1	3 ns
10	`tcp_xmit()`	no	24–28	100% local L1	3 ns
25	`dev_xmit()`	yes	24–28	95% foreign cache	200 ns
50	`kfree()`	no	0–128	—	0

Data Profile

- Answers the question *“Who are the bad guys?”*
- Ranked list of data types (structs) showing which ones cause the most expensive memory accesses
- High-level view to find the top 2-3 “hot” data types; then you drill down into those types using the other views

Data type	**% of cache misses**	**Avg cost**	**Cross-core bouncing?**
`struct tcp_sock`	35%	high	yes
`struct skbuff`	22%	high	yes
`struct foo`	5%	low	no

Tabular representation, the actual paper has more detailed numbers, but this is the idea

Miss Classification

- Answers the question *"Why are they bad?"*
- Breaks down, for one data type, why its accesses are slow (too much data, bad layout, or cross-core sharing)
  - capacity = too much stuff
  - conflict = unlucky placement
  - sharing = too many cores touching the same line

Miss type	**% misses for `tcp_sock`**	**Intuition (in plain words)**
Capacity	55%	Too many `tcp_sock` objects alive to fit in cache
Conflict	10%	Layout causes them to fight for the same cache slots
True sharing	20%	Multiple cores read/write the same fields in this struct
False sharing	15%	Different cores touch different fields on same cache line

Simplified tabular representation for `struct tcp_sock`

Working Set

- Answers the question *"How many of them are in play at once?"*
- Estimates how many objects of each type are resident in cache and in which associativity sets

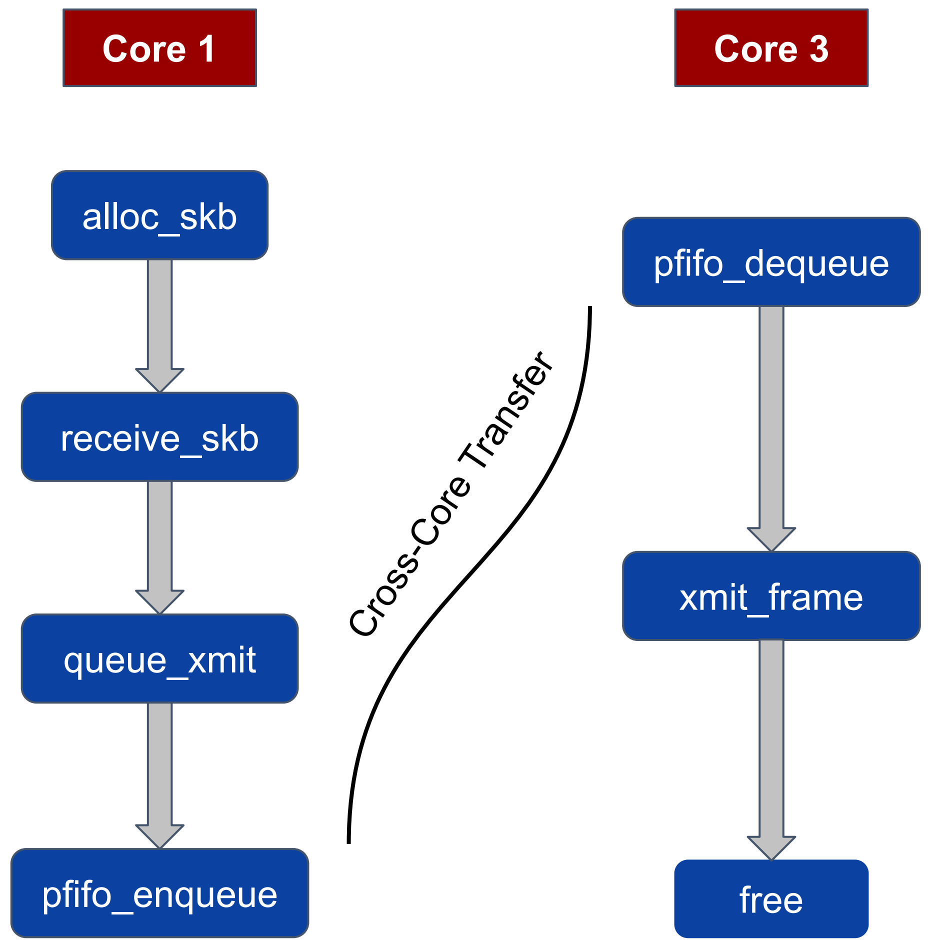
Example Interpretation:

- At moderate load, you might see ~1 MB of `tcp_sock` in the working set → OK.
- At high load, ~11 MB of `tcp_sock` show up in the working set → exceeding what the cache can handle efficiently → capacity misses skyrocket.

Load level	**Approx. working set size** (`tcp_sock`)
Moderate load	~1 MB
High load	~11 MB

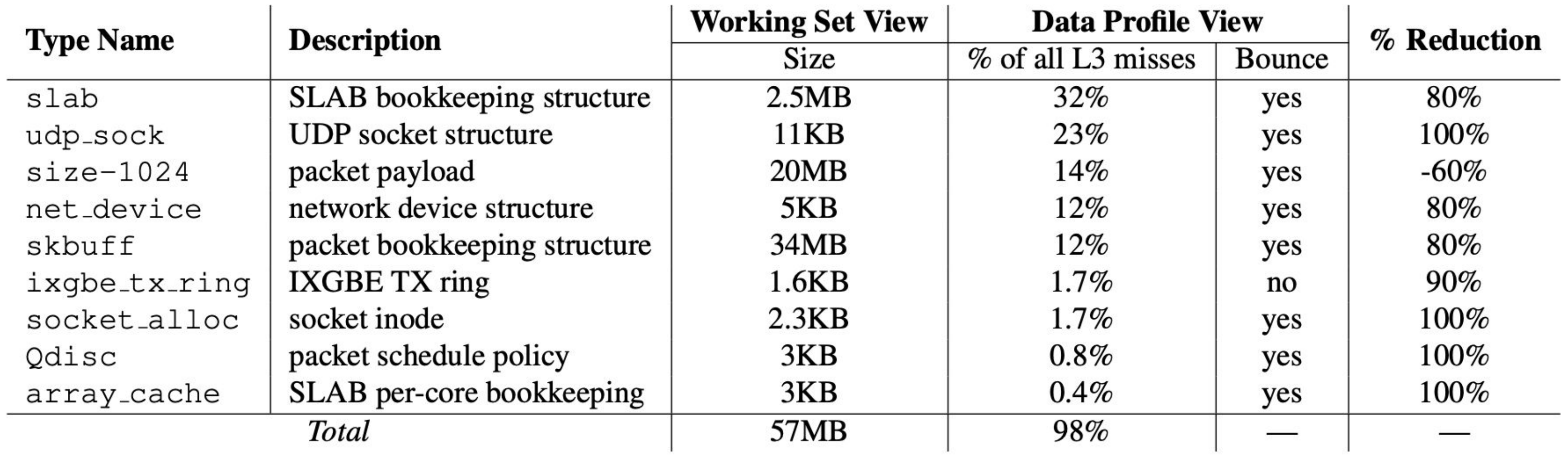
Data Flow

- Answers the question *"What happens during their life?"*
- "Life Story" diagram for a typical object: which code/threads/cores touch it, in what order, and where the costly accesses and cross-core moves happen.



Results: Memcached

- Memcached didn’t scale even with one instance per core
- DProf showed a few data types caused most cache misses
- Packet buffers (1024-byte) and `skbuff` moved across cores
- Cross-core movement means costly “true sharing” misses

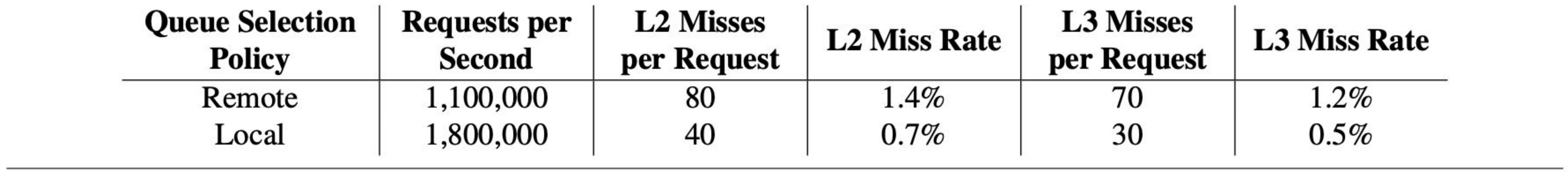


Type Name	Description	Working Set View	Data Profile View		% Reduction
		Size	% of all L3 misses	Bounce	
`slab`	SLAB bookkeeping structure	2.5MB	32%	yes	80%
`udp_sock`	UDP socket structure	11KB	23%	yes	100%
`size-1024`	packet payload	20MB	14%	yes	-60%
`net_device`	network device structure	5KB	12%	yes	80%
`skbuff`	packet bookkeeping structure	34MB	12%	yes	80%
`ixgbe_tx_ring`	IXGBE TX ring	1.6KB	1.7%	no	90%
`socket_alloc`	socket inode	2.3KB	1.7%	yes	100%
`Qdisc`	packet schedule policy	3KB	0.8%	yes	100%
`array_cache`	SLAB per-core bookkeeping	3KB	0.4%	yes	100%
	Total	57MB	98%	—	—

Table 4. Working set and data profile views for the top data types in memcached as reported by DProf. The percent reduction column shows the reduction in L3 misses caused by changing from a remote to a local queue selection policy. The number of L3 misses for `size-1024` increased with the local policy.

Memcached: Fix + Improvement

- Cause: packets placed on non-local transmit queues
- Means packets were processed by multiple cores
- Fix: choose a queue local to the current core
- Result: much fewer L2/L3 misses + ~57% higher throughput

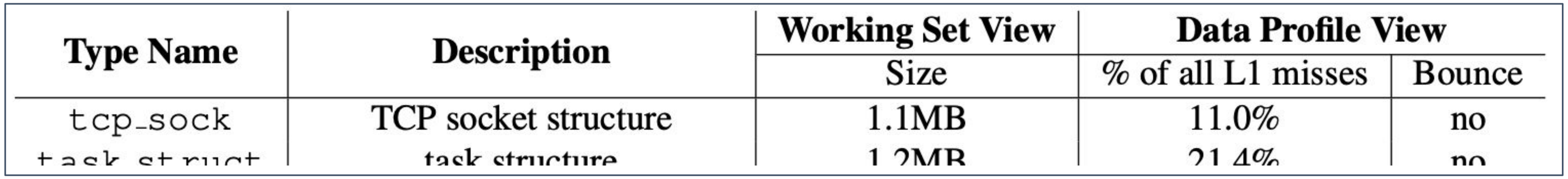


Queue Selection Policy	Requests per Second	L2 Misses per Request	L2 Miss Rate	L3 Misses per Request	L3 Miss Rate
Remote	1,100,000	80	1.4%	70	1.2%
Local	1,800,000	40	0.7%	30	0.5%

Table 5. Cache misses per request and cache miss rates for the L2 and L3 caches. Results are shown running memcached for both transmit queue selection policies. These numbers were gathered using hardware performance counters to collect cache miss and data access counts.

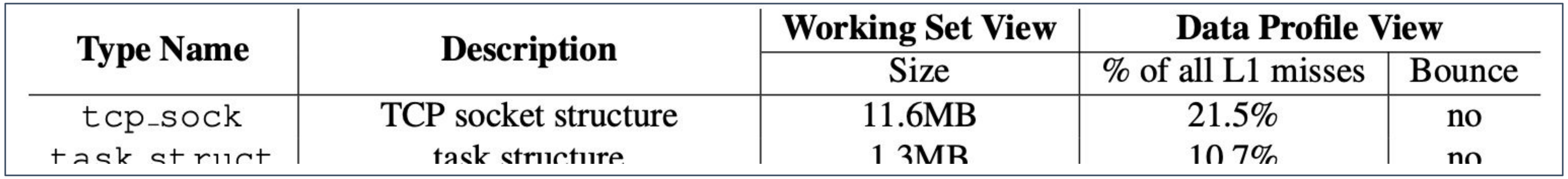
Results: Apache

- Apache throughput dropped at high load
- DProf compared "peak" vs "drop-off" runs
- `tcp_sock` objects became the main source of misses
- Their working set grew from ~1 MB -> ~11 MB



Type Name	Description	Working Set View	Data Profile View	
		Size	% of all L1 misses	Bounce
tcp_sock	TCP socket structure	1.1MB	11.0%	no
task_struct	task structure	1.2MB	21.4%	no

Table 7. Working set and data profile views for the top data types in Apache at peak performance as reported by DProf.



Type Name	Description	Working Set View	Data Profile View	
		Size	% of all L1 misses	Bounce
tcp_sock	TCP socket structure	11.6MB	21.5%	no
task_struct	task structure	1.3MB	10.7%	no

Table 8. Working set and data profile views for the top data types in Apache at drop off as reported by DProf.

Apache: Fix + Improvement

- Cause: too many waiting TCP connections
- Long waits pushed `tcp_sock` objects out of cache
- Fix: limit the accept queue (admission control)
- Result: no more drop-off + ~16% higher performance

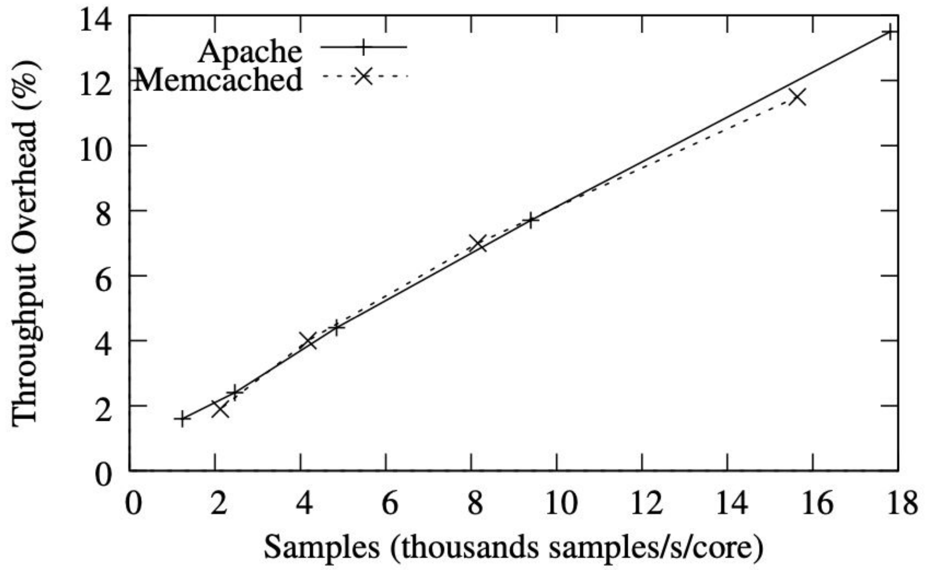


Figure 3. DProf overhead for different IBS sampling rates. The overhead is measured in percent connection throughput reduction for the Apache and memcached applications.

Our Reflections: Strengths

1. **Data-Centric Perspective Instead of Code-Centric Profiling:** Traditional profilers attribute costs to functions or instructions, which can hide cache-related performance issues when misses are spread thinly across code paths.

2. **Can Provide Clear Fixes:** DProf doesn't just report that cache misses are happening — it helps identify why (e.g., true sharing, capacity, conflict), making it easier to choose the correct optimization strategy.

Our Reflections: Limitations

1. **High Profiling Overhead:** DProf incurs significant runtime and memory overhead due to frequent interrupts and large sample storage.
2. **Hardware Dependency and Limited Visibility:** The tool depends heavily on AMD IBS and a small number of debug registers. Only four registers can trace memory at a time, forcing pairwise sampling and making fine-grained tracking difficult. Lack of direct cache inspection also forces indirect inference about working set size.
3. **Statistical sampling needs steady workloads:** As a statistical sampler, DProf requires workloads that are stable over time. Irregular or phase-shifting applications may not produce representative traces, reducing its accuracy and usefulness.