# Playing Fetch with CAT

## Composing Cache Partitioning and Prefetching for Task-based Query Processing

Qitian Zeng
Illinois Institute of Technology
qzeng3@hawk.iit.edu

Kyle C. Hale
Illinois Institute of Technology
khale@cs.iit.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

## ABSTRACT

Software prefetching and hardware-based cache allocation techniques (CAT) have been successfully applied in main-memory database engines to fetch data into cache before it is needed and to partition a shared last-level cache (LLC) to prevent concurrent tasks from evicting each others' data. We investigate the interaction of these techniques and demonstrate that while a single prefetching strategy is sufficient, the combination of both techniques is only effective if the cache partitioning strategy adapts the partitioning based on the types of tasks currently sharing an LLC. We present a simple, yet effective, scheme that uses prefetching and adapts cache partition allocations dynamically.

**ACM Reference Format:**
Qitian Zeng, Kyle C. Hale, and Boris Glavic. 2021. Playing Fetch with CAT: Composing Cache Partitioning and Prefetching for Task-based Query Processing. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3465998.3466016

## 1 INTRODUCTION

Memory access latency and memory bandwidth limits are major bottlenecks for in-memory databases that use (parallel) operator implementations that may consume data at a rate that exceeds the bandwidth of the memory bus. Smart caching techniques can mitigate these bottlenecks to some degree. However, not all tasks within a database engine benefit equally from the cache. Operators like column scans exhibit no temporal locality of memory access (*non-temporal tasks*), because they access memory locations exactly once while scanning through their input. Similarly, tasks with low-degree of temporal locality of memory access (*low-degree temporal tasks*) such as a hash aggregate or join with a large hash table, access the same memory location repeatedly, but the average duration between accesses is high. Non-temporal and low temporal operators do benefit from prefetching (loading data into cache before it is accessed), but given the limited size of the cache, an accessed memory location is likely to be evicted from cache before it will be accessed again. In contrast, operators with high temporal locality of memory access (*high-degree temporal tasks*) such as a hash aggregate with a small hash table (a small number of groups) frequently access a relatively small number of memory locations. Thus, such operators do benefit from caching significantly.

When concurrent tasks run on cpus that share a LLC, non-temporal and low-degree temporal tasks will interfere and pollute the cache for high-degree temporal tasks. Cache contention and interference are common in in-memory databases [16, 21] and in other contexts [3, 11, 23, 29] such as datacenter scheduling [9], HPC [10, 15], and mixed interactive/batch workload scheduling [6].

To improve the cache utilization for a task, we should try to ensure the data that the task will access in the near future is (i) loaded into cache before the task accesses the data and (ii) is still available when the task accesses the data. Software prefetching[2, 4, 14, 18–20] gives us control over (i) by allowing us to read data into cache preemptively at the cost of increased bandwidth usage if we mispredict the access pattern of a task or if the prefetched data has been evicted from the cache before it has been used. Partitioning of a shared LLC enables us to control (ii) since tasks in different partitions cannot evict each other's data from the cache.

Software prefetching has long been supported by hardware, usually via prefetch *hint* instructions. Compilers have been able to automatically insert software prefetching hints to improve indirect memory access latency [2, 18, 20]. In the database community, different prefetching techniques such as *Group Prefetching* [4] and *Asynchronous Memory Access Chaining* [14] were introduced to address the multi-step dependent memory access pattern in database operators. Combined with query compilation and vectorization these techniques significantly improve performance [19].

The use of LLC partitioning for reducing cache pollution has also been studied in prior work. Intel's Cache Allocation Technology (CAT) [8] makes LLC partitioning straightforward. Before CAT was introduced, partitioning was achieved


*DAMON'21, June 20–25, 2021, Virtual Event, China*

ACM ISBN 978-1-4503-8556-5/21/06.
https://doi.org/10.1145/3465998.3466016

through way-partitioning [5, 22], set-partitioning [22, 26], and through replacement policies [12, 25, 27, 28] which all need different levels of additional hardware support, with the exception of one applicable (software-only) technique using page coloring [17, 24]. Lee et al. [16] applied page-coloring in PostgreSQL, along with query optimization and scheduling to minimize cache conflicts. However, page coloring does not provide fine-grained control, and it requires OS support.

Noll et al. [21] discuss temporality of memory access for database operators and experimentally study the impact of CAT on concurrent execution of database queries. This study has shown that per query LLC partitioning can be effective. By restricting the LLC size of queries with non-temporal access patterns (like column scans), the throughput of queries with high degrees of temporal locality (like hash aggregate) is increased by up to 50%. However, this strategy is static. Each query is assigned an LLC partition based on a heuristic evaluation whether it is cache sensitive or not, and that partition is in effect until the query completes.

In summary, while cache partitioning and prefetching have both been studied extensively in isolation, the combination of these techniques, perhaps surprisingly, has to the best of our knowledge not been studied in the database context. We explore this question empirically and demonstrate that static LLC partitioning schemes are less beneficial when combined with prefetching. In fact, when multiple temporal tasks are running on the same LLC it may be better to just use prefetching alone! However, not all is lost. We propose a *dynamic partitioning and prefetching scheme* that adapts cache allocations based on what tasks are concurrently using the same LLC. This is feasible in the databases, because the query processing engine has *a priori* information about the access patterns of tasks. With dynamic partitioning, we can achieve moderate improvements over prefetching and prefetching combined with static partitioning.

## 2 COMPOSING CAT AND PREFETCHING

Using hash aggregate and scan as representative examples of temporal and non-temporal tasks, we seek to answer the following questions empirically:

- Does the composition of software prefetching and LLC partitioning produce performance benefits for competing tasks?
- Is a static LLC partitioning scheme sufficient?

As alluded to in the introduction, the access patterns of database tasks can broadly be classified into two categories based on their memory access: (i) sequential access only, which is non-temporal and benefits neither from software prefetching[1] nor from having more LLC cache (we will use NT to denote this class of tasks). This class of tasks includes sequential scans; (ii) random access (which we denote by **T**) which is temporal. Examples of this type of task are hash aggregation and the hash joins. As discussed earlier, the degree of temporal locality may vary based on factors such as the hashtable size of a hash aggregate. For our evaluation we use microbenchmarks of concurrent T-NT (scan), and T-T tasks (aggregate). However, the results generalize to other operator implementations (or parts thereof) as long as they fall into these categories.

### 2.1 Experimental Setup

Next we describe our task model, the database schema used in the experiments, and our experimental testbed. All tables in our experiments follow the schema shown below, using a columnar storage model. Each table is divided into chunks consisting of a number of rows. Within each chunk, all values of a column are stored in a contiguous area of memory. All attribute values are 32 bit integers.

*Task 1: Column Scan.* This task scans through a 50GB single-column table `X(A)`. this roughly corresponds to the following SQL query: `SELECT * FROM X;` Due to the limited physical memory in our testbed (48GB), this is performed by scanning through a 1GB chunk 50 times. The column scan exclusively performs sequential memory accesses and acts as an LLC polluter and memory bandwidth hog.

*Tasks 2 & 3: Hash Aggregate.* These tasks perform the same operation as the query: `SELECT SUM(B) GROUP BY A FROM Y;` over a table `Y(A,B)`. Here, `Y` has $2^{28} =\sim 268M$ rows, making both columns 1GB in size. The difference is the number of unique values of column A. In Task 2, values are randomly chosen from $100 * 2^{14}$ distinct values, while in Task 3 they are chosen from $100 * 2^{15}$ distinct values. With one hash entry taking up 12 bytes and a 75% fill ratio, Task 2 needs a 25MB hash table and Task 3 needs 50MB. We use xxHash [1] as the hash function with linear probing to handle collisions. These two tasks perform random memory access to a small and large region of memory, respectively. Like many database operations, hash aggregate has indirect memory access patterns and thus can benefit from software prefetching.

*Experimental Testbed.* We evaluate our microbenchmarks on a 10-core Intel Xeon E5-2630v4 (Broadwell) @ 2.4GHz with 48GB DDR4 2133Mhz RAM. Broadwell incorporates an on-die memory controller with a maximum bandwidth of

[1]Sequential access benefits significantly from prefetching. However, because of its predictable access pattern, hardware prefetching is quite effective and no additional benefits are gained by software prefetching.

| Cache | Size | Associativity | Shared? | Policy |
|---|---|---|---|---|
| L1I | 32KB | 8-way SA | No | WB |
| L1D | 32KB | 8-way SA | No | WB |
| L2 | 2.5MB | 8-way SA | No | WB |
| L3 | 25MB | 20-way SA | Yes | WB |

**Table 1: Cache organization for our testbed machine**

64GB/s. The cache organization is shown in Tab. 1. Hyperthreading (SMT) and frequency scaling are disabled[2]. Our testbed machine runs Fedora Server 31 with Linux kernel v5.6.18. Huge page support is enabled, so all chunks are backed by 1GB huge pages, limiting TLB pressure and mitigating caching contention from the hardware page walker. Our code is compiled with gcc 9.3.1 and optimization level -O3. For each experiment, tasks are instrumented for performance measurements using the Linux `perf` API; we measure task execution times using the cpu-clock count as reported by perf. Execution times are normalized to a 0-1 range, with 1 being the highest value in the plot. Unless otherwise stated, experimental results are reported as the arithmetic mean of 20 repetitions with error bars depicting 95% confidence intervals. For experiments with software prefetching, we use strided prefetching, where the stride is established by prefetching entries that will be accessed in subsequent loop iterations (prefetch hash entry for tuple $n + k$ when processing tuple $n$ for stride $k$). We did ran experiments for many different strides. We observed that performance benefits saturate at $k = 64$. Thus, we report a fixed stride of 64 in this paper for brevity. In general, collisions in hash tables can reduce the effectiveness of prefetching. However, note that we use linear probing [13] and the smallest unit of data that can be cached is 64 bytes. Thus, when the bucket for the next key we are going to access in the hashtable is prefetched then with high probability any bucket that we may access during linear probing will belong to the same 64 byte chunk and, thus, is already present in the cache.

Our three tasks represent different LLC sensitivities given the $25MB$ LLC size of our testbed. Task 1 is LLC *insensitive* since its memory access is sequential—so that it can benefit from the hardware prefetcher—and non-temporal—so that leaving data in cache does not improve hit rate. Task 2 is highly LLC *sensitive* since it can achieve up to a 100% hit rate when its full hash table is cache resident. Task 3 is less LLC sensitive since even when given the entire LLC it has a hit rate of only 50% (because the result of hashing is essentially random for a reasonable choice of hash function and data that is not skewed).

[2]The BIOS is configured for the "maximum performance profile," thus disabling DVFS.

## 2.2 Static Partitioning Strategy

Before evaluating cache partitioning and prefetching, we first develop a partitioning strategy that is optimal under reasonable assumptions. As we shall see, a greedy approach best maximizes LLC efficiency. In developing our formalism, we make the following simplifying assumptions:

(1) The workload begins with a hot cache (cold misses are not considered).
(2) The cache references frequency of all tasks is the same independent of their co-location with other tasks.
(3) We assume that given two distinct tasks $t_x$ and $t_y$ accessing memory randomly, the cache lines they access are distinct. That is, if $R(t)$ is the set of all cache lines accessed by task $t$, then $R(t_x) \cap R(t_y) = \emptyset$.

Let $n$ be the number of concurrently executing tasks[3]. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a partitioning where $p_i$ is the size of the LLC partition (in MB) for task $t_i$ and let $M$ be the total LLC capacity of the system. Then assumption 2 allows us to measure the cache efficiency of a single task $t_i$ as $\min(1, \frac{p_i}{d_i})$. Let $d_1, d_2, ..., d_n$ be the data size in MB that task $t_i$ performs random access to, i.e. the ideal LLC partition size given a sufficiently large cache. We assume $d_1 \leq d_2 \leq \ldots \leq d_n$ WLOG. We define $f(P) : \mathbb{N}^n \rightarrow \mathbb{R}$, the total achieved cache efficiency given some partitioning $P$ as

$$f(P) = \sum_{i=1}^{n} \min(1, \frac{p_i}{d_i})$$

with $\sum_{i=1}^{n} d_i > M$ (otherwise the optimal strategy is to set $p_i = d_i$ for all tasks). The optimization problem we want to solve is to choose the partitioning $P$ that maximizes $f(P)$:

$$\underset{P}{\operatorname{argmax}} f(P) \quad \textbf{subject to} \quad \sum_{i=1}^{n} p_i \leq M$$

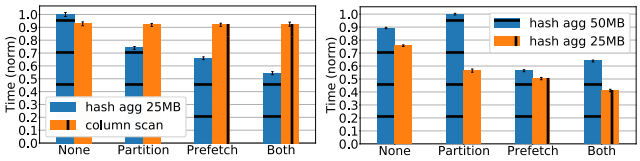
This is a fractional knapsack problem, and the following greedy algorithm has been proven to be optimal [7]:

```
1  for i = 1 to n
2      p_i = min(M, d_i)
3      M = M − p_i
```

For our example we have $n = 3$, $d_1 = 0$, $d_2 = 25$, and $d_3 = 50$. This means we always set $p_1 = 0$. When Task 2 is concurrent with Task 3, we set $p_2 = 25$ and $p_3 = 0$.

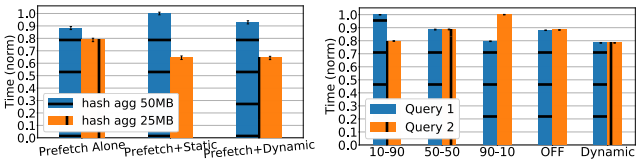
*LLC Partitioning with CAT.* CAT provides multiple Classes of Service (CLOS), each associated with a bitmask that defines which parts of the cache are available to this CLOS. An application's LLC access is controlled by assigning it to a CLOS. Intel's CAT implementation does not presently allow for zero-sized partitions; We thus reserve a minimal 10%

[3]Here we use *concurrently* to mean tasks running on distinct cores sharing the same LLC *at the same time*.




(a) Scan and hash aggregate (b) Two hash aggregates

**Figure 1: Combining prefetching and LLC partitioning.**




(a) Workload from Fig. 1b with dynamic partioning (b) mixed T & NT task queries, prefetching enabled

**Figure 2: Static vs. dynamic partitioning**

partition to be shared among all tasks that receive a partition of size zero from the algorithm.

## 2.3 Sensitivity Analysis

We now study the performance impact of CAT and prefetching for various combinations of the tasks described above.

*Concurrent Column Scan & Hash Aggregate.* We first investigate a column scan (non-temporal access) running concurrently with a hash aggregate task (temporal access). Here, prefetching is only applied for the hash aggregate. As mentioned before we fix the prefetch stride for the hash aggregate to 64 entries. Since Task 1's sequential access is amenable to hardware prefetching and thus does not require LLC, increasing its partition size results in unnecessary pollution. We thus partition the LLC to restrict Task 1 to a partition of 10% giving Task 2 free rein (100%). Fig. 1a shows that the combination of prefetching and partitioning has a positive impact on performance for this scenario. However, at the presence of prefetching, the speedup of cache partitioning is reduced from 26% to 17%.

**Observation:** *Prefetching limits the impact of partitioning.*

*Concurrent Hash Aggregates (Task 2 & Task 3).* For this workload (two hash aggregates), we apply prefetching for both tasks. We partition the LLC by prioritizing tasks with higher LLC sensitivity (temporal access). Here, we limit Task 3 to 10% and allot 100% to Task 2. As Fig. 1b shows, without software prefetching, LLC partitioning achieved a 26% speedup for Task 2 while the slowdown of Task 3 is less than 10%; with software prefetching enabled, however, the effect of LLC partitioning is significantly reduced. The speedup of Task 2 falls to 18%, and the slowdown of Task 3 increases to 13%; the absolute speedup of Task 2 is roughly on par with the slowdown of Task 3. Software prefetching, however, always has a positive effect with or without LLC partitioning.

**Observation:** *For concurrent temporal tasks, the benefits of partitioning are further reduced when prefetching is used. Even worse, partitioning amplifies runtime differences between tasks.*

## 2.4 Discussion

Based on these results, we conclude that, assuming an appropriate choice of stride, software prefetching is always beneficial with or without LLC partitioning. Conversely, static LLC partitioning does *not* always compose well with software prefetching. The effect is in general limited and can be negative. To cash in this limited positive effect, we need a dynamic yet simple strategy, because the narrow margin does not allow for a complex or expensive strategy.

# 3 DYNAMIC PARTITIONING

We now improve our greedy LLC partitioning strategy by dynamically re-partitioning (over time) to adapt to changes in the mix of tasks sharing an LLC. We begin by reexamining Fig. 1b where the static partitioning strategy was ineffective. Note the gap between the execution time for the two tasks after enabling LLC partitioning. Even after Task 2 finishes, Task 3 still runs with only 10% of the LLC for a significant period of time. An obvious improvement is to allow Task 3 to have the entire LLC as soon as Task 2 completes. A simple dynamic partitioning strategy is to adjust the partitioning whenever a task begins or finishes using our greedy strategy.

We evaluate this algorithm using the microbenchmark from Fig. 1b. The result is shown in Fig. 2a. Note the speed-up for the hash aggregate with 50MB hash table (task 3). While this speed-up is small, cache partitioning now has a net positive impact. We also evaluated more complex queries. For this experiment we use two queries: Query 1 that runs a 30GB column scan and then performs Task 2 (a 25MB hash aggregate) and Query 2 executes the same tasks in reverse order. As Fig. 2b shows, static partitioning with prefetching (we tested $10\% - 90\%$, $50\% - 50\%$ and $90\% - 10\%$ allocations) does not benefit over prefetching (`OFF`). However, dynamic partitioning with prefetching improves performance for both queries (11%), because both queries benefit from having the entire LLC for their temporal tasks.

# 4 CONCLUSION

CAT and software prefetching are effective techniques for improving cache utilization in main-memory database systems. However, static partitioning is not always beneficial when prefetching is applied. Thus, software prefetching and LLC partitioning *must be composed carefully*. We present a simple, yet effective, dynamic partitioning strategy that pays off for mixed, task-based query processing workloads. In short, *"If you want to play fetch with CAT, you need a dynamic CAT."*

## REFERENCES


[1] [n.d.]. xxHash - Extremely fast non-cryptographic hash algorithm. https://cyan4973.github.io/xxHash/

[2] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *CGO*. 305–317.

[3] Liuhua Chen, Haiying Shen, and Stephen Platt. 2016. Cache contention aware Virtual Machine placement and migration in cloud datacenters. In *ICNP*. 1–10.

[4] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *TODS* 32, 3 (2007), 17–es.

[5] D. Chiou, L. Rudolph, S. Devadas, and B. Ang. 2000. Dynamic Cache Partitioning via Columnization. In *DAC*.

[6] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. In *ISCA*. 308–319.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

[8] Intel Corporation. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *White Paper* (2015).

[9] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS*. 77–88.

[10] Jack Dongarra, Bernard Tourancheau, Guillaume Aupy, Anne Benoit, Brice Goglin, Loïc Pottier, and Yves Robert. 2019. Co-Scheduling HPC Workloads on Cache-Partitioned CMP Platforms. *Int. J. High Perform. Comput. Appl.* 33, 6 (2019), 1221–1239.

[11] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. *SIGPLAN Notes* 51, 6 (2016), 251–265.

[12] Ravi R. Iyer. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS*, Paul Feautrier, James R. Goodman, and André Seznec (Eds.). 257–266.

[13] Don Knuth. 1963. Notes On "Open" Addressing.

[14] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *PVLDB* 9, 4 (2015), 252–263.

[15] Brian Kocoloski, Yuyu Zhou, Bruce Childers, and John Lange. 2015. Implications of Memory Interference for Composed HPC Applications. In *MEMSYS*. 95–97.

[16] Rubao Lee, Xiaoning Ding, Feng Chen, Qingda Lu, and Xiaodong Zhang. 2009. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *PVLDB* 2, 1 (2009), 373–384.

[17] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*. 367–378.

[18] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *ASPLOS*. 222–233.

[19] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *PVLDB* 11, 1 (2017), 1–13.

[20] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS*. 62–73.

[21] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2018. Accelerating Concurrent Workloads with CPU Cache Partitioning. In *ICDE*. 437–448.

[22] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. 2000. Reconfigurable caches and their application to media processing. In *ISCA*, Alan D. Berenbaum and Joel S. Emer (Eds.). 214–224.

[23] Bharath Ravi, Hrishikesh Amur, and Karsten Schwan. 2013. A-Cache: Resolving cache interference for distributed storage with mixed workloads. In *CLUSTER*.

[24] Livio Soares, David K. Tam, and Michael Stumm. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *MICRO*. 258–269.

[25] Daniel Sánchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *ISCA*, Ravi Iyer, Qing Yang, and Antonio González (Eds.). 57–68.

[26] Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Bharadwaj Amrutur, Ravi R. Iyer, Srihari Makineni, and Donald Newell. 2006. Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions. In *MICRO*. 433–442.

[27] Carole-Jean Wu and Margaret Martonosi. 2011. Adaptive timekeeping replacement: Fine-grained capacity management for shared CMP caches. *ACM Trans. Archit. Code Optim.* 8, 1 (2011), 3:1–3:26.

[28] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*. 174–183.

[29] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic Cache Contention Detection in Multi-Threaded Applications. In *VEE*. 27–38.