PACMan: Coordinated Memory Caching for Parallel Jobs

Ganesh Ananthanarayanan[1], Ali Ghodsi[1], Andrew Wang[1], Dhruba Borthakur[2], Srikanth Kandula[3]
Scott Shenker[1], Ion Stoica[1]

[1] University of California, Berkeley [2] Facebook [3] Microsoft Research

Draft under review, please do not redistribute.

Abstract

Data-intensive analytics, operating on large clusters, is an important driver of modern Internet services. With trends of large memories in these clusters, in-memory caching of inputs is an effective way to reduce completion times of these analytical jobs. The key challenge, however, is that these jobs run multiple tasks in *parallel* and a job is sped up only when inputs of all such parallel tasks are cached. We build PACMan, a caching service that *coordinates access to the distributed caches*. Coordinated cache replacement is a key aspect of PACMan; hit-ratios do not necessarily improve job completion times. Three production traces – from Facebook, Bing and Yahoo! – show a heavy-tailed distribution of job sizes. Small interactive jobs dominate by count while large production jobs consume the most resources. We implement two cache replacement policies on top of PACMan's coordinated infrastructure – LIFE that minimizes average completion time by evicting large incomplete inputs, and SIFE that speeds up large jobs by evicting small incomplete inputs. Evaluations show that we reduce job completion times by up to 53% and improve cluster utilization by up to 52%. LIFE and SIFE beat their competitors in both job completion and utilization.

1 Introduction

Cluster computing has become a major platform, powering both large Internet services and a growing number of scientific applications. Data-intensive frameworks, such as MapReduce [14] and Dryad [21], allow users to perform data mining and sophisticated analytics, automatically scaling up to thousands of machines.

A noteworthy development in datacenters has been the change in memory capacity of the machines. With falling prices, machines have more memory than ever before (*e.g.,* 64GB per machine is not uncommon), large shares of which are unutilized; the median and 95^{th} percentile utilizations in the Facebook cluster are 19% and 42%, respectively. In light of this trend, we investigate the use of *memory-locality* to speed-up jobs in data-intensive clusters by caching their input data.

Data-intensive jobs, typically, have a phase where they process the input data (*e.g., map* in MapReduce [14], *extract* in Dryad [21]). This phase simply reads the raw input and writes out parsed output to be consumed for further computations. Naturally, this phase of the job is IO-intensive. Workloads from Facebook, Microsoft Bing and Yahoo! datacenters, consisting of thousands of servers, show that this IO-intensive phase constitutes 79% of a job's duration and consumes 69% of its resources. Data is typically accessed by multiple jobs; only 6% of jobs read singly-accessed data. Our proposal is to speed up these IO-intensive phases, by caching their input data in memory. Data is cached after the first access thereby speeding up subsequent accesses. Using memory caching to improve performance has a long history in computer systems, *e.g.,* [5, 15, 17, 19]. We argue, however, that the *parallel* nature of data-intensive jobs differentiates them from previous systems.

Frameworks split jobs in to multiple *tasks* that are run in parallel. There are often enough idle compute slots for small jobs, consisting of few tasks, to run all their tasks in parallel. Such tasks start at roughly the same time and run in a single *wave*. In contrast, large jobs, consisting of many tasks, seldom find enough compute slots to run all their tasks at the same time. Thus, only a subset of their tasks, equal to the number of compute slots, run in parallel. As and when tasks finish and vacate slots, new tasks get scheduled on them[1]. We define the number of parallel tasks as the *wave-width* of the job.

A characteristic of all workloads we have analyzed is that small jobs are single-waved, and, in addition, tend to have *much smaller* wave widths than larger jobs. This is because they have fewer tasks than the average number of slots the cluster scheduler typically allocates to a job.

The wave-based execution implies that small single-waved jobs require full memory-locality – an *all-or-nothing* property – to improve their completion time. They run all their tasks in one wave and their completion time is proportional to the duration of the longest task. Large jobs, on the other hand, improve their completion time with every wave-width of their input being cached. Note that we do not care about the exact set of tasks that run in a wave, we only care about the wave-width, *i.e.,* how many of them run simultaneously.

Our position is that *coordinated management* of the caches is required to ensure that enough tasks of a parallel job have memory-locality to improve their completion time. Coordination provides a global view that can

[1] We use the terms "small" and "large" jobs to refer to their input size. The number of tasks in a job is proportional to its input size.

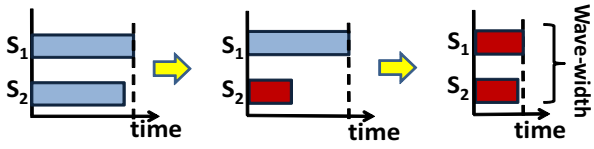
be used to decide what to evict from the cache, as well as where to place tasks so that they get memory-locality. To this end, we have developed PACMan – Parallel All-or-nothing Cache MANager – an in-memory caching system for parallel jobs. On top of PACMan's coordinated infrastructure, appropriate placement and eviction policies can be implemented to speed-up parallel jobs.

One such coordinated eviction policy we built, LIFE, aims to *minimize the average completion time of jobs* in the cluster. In a nutshell, LIFE favors small jobs and replaces the cached blocks of the largest incomplete file. The design of LIFE is driven by two observations. First, a small job requires to cache less data than a large job to get the same decrease in completion time. This is because the amount of cache required by a job is proportional to its wave-width, and, as mentioned earlier, small jobs have smaller wave-widths. Second, since small jobs tend to be single-waved, we need to cache their entire inputs to improve their completion times. This also means that we need to retain their entire inputs in cache, hence the heuristic of replacing blocks from incomplete files.

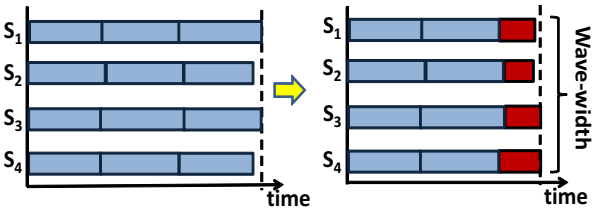
Note that maximizing cache hit-ratio – the metric of choice of traditional replacement policies – does not necessarily minimize the average completion time, as it ignores the wave-width constraint of parallel jobs. Indeed, consider a simple workload consisting of 10 equally-sized single-waved jobs. A policy that caches only the inputs of five jobs will provide a better average completion time, than a policy that caches 90% of the inputs of each job, which will not provide any completion time improvement over the case in which no inputs are cached. However, the first policy will achieve only 50% hit-ratio, compared to 90% hit-ratio for the second policy.

In addition to LIFE, we implemented another eviction scheme, SIFE, which aims to improve the completion times of large jobs. This policy is motivated by the fact that some of the largest jobs are production jobs that generate results (*e.g.,* search indexes, social graphs) critical for front-end applications. SIFE is symmetric to LIFE: instead of replacing cached blocks from the largest incomplete files, it replaces cached blocks from the *smallest* incomplete files. An unexpected side-effect of SIFE is that it has a higher hit-ratio than all other policies we tried, thereby leading to *efficient cluster utilization*. This is because of the correlation in these clusters where input files of large jobs are accessed more often.

We make the following contributions. First, we identify the need for and build a coordinated caching framework for parallel jobs. Second, we perform a detailed quantification of workloads from large clusters at Facebook, Bing and Yahoo! to understand the feasibility and potential of caching. Finally, we provide two cache replacement policies – LIFE and SIFE – that optimize for job completion time instead of cache hit-ratio.



(a) Single-wave job. Wave-width = 2.



(b) Multi-wave job. Wave-width = 4.

Figure 1: **Single-wave and multi-wave examples. Memory-local tasks are dark blocks. Completion times (dotted line) reduce only when wave-widths of data are cached.**

We have integrated PACMan with Hadoop HDFS [2]. PACMan is evaluated by running workloads from data-centers at Facebook and Bing on our EC2 cluster. We show that overall job completion times reduce by up to 53% with LIFE and cluster utilization improves by up to 52% with SIFE. Notably, completion times of small jobs reduce by 77% with LIFE and large jobs by up to 50% with SIFE. In speeding up jobs, LIFE outperforms traditional schemes like LRU, LFU, and even MIN [11], which is provably optimal for hit-ratios. In addition to speeding up large jobs, SIFE's hit-ratio is second only to MIN thereby significantly improving cluster efficiency.

2 Impact of Wave-width

In this section, we build intuition on the impact of caching wave-widths of input on a job, and then proceed to understand its effect on a cluster of jobs.

Achieving memory-locality for a task will shorten its completion time. By itself, this need not speed up the job. *Jobs speed up when an entire "wave-width" of input is cached* [2]. Wave-width of a job is defined as the number of simultaneously executing tasks. Therefore, jobs that consist of a single wave need 100% memory-locality – all-or-nothing – to reduce their completion time, whereas jobs consisting of many waves will benefit as we incrementally cache inputs in multiples of their wave-width. Figure 1 highlights this phenomenon on a 4-slot cluster. The small job runs both its task simultaneously and will speed up only if both get memory-locality. The large job,

[2]This assumes similar task durations, which turns out to be true in practice. The 95^{th} percentile of the coefficient-of-variance ($\frac{stdev}{mean}$) among tasks in the data-processing phase (*e.g.,* map) of jobs is 0.08.

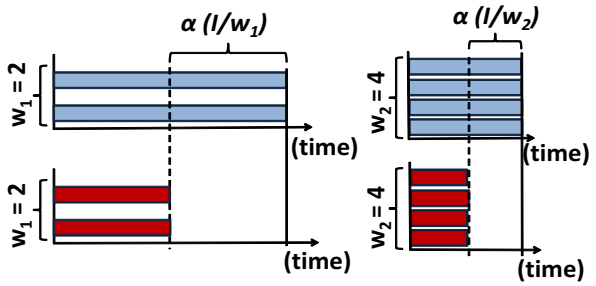


Figure 2: **Comparison of savings in completion time. Savings decrease as wave-width increases. Both examples read I amount of data. Solid lines represent the completion times without caching, dotted lines with caching.**

on the other hand, consists of 12 tasks and can run 4 of them at a time. Hence, its completion time improves in steps when 4, 8 and 12 tasks run memory-locally [3].

Let us now move to a cluster with multiple jobs. How do we allocate the available cache space so that we improve the average job completion time? Assume all jobs in the cluster are single-waved. Every job j has a wave-width of w and input size of I. Let us assume the input is equally distributed among its tasks. Each task's input size is $\left(\frac{I}{w}\right)$ and its duration is proportional to its input size. Memory-locality reduces its duration by α. The factor α is dictated by the difference between memory and disk throughputs, as well as additional overheads, such as data deserialization and decompression.

To speed up this singly-waved job, we need I units of cache space. On spending I units of cache space, the saving in completion time would be $\alpha\left(\frac{I}{w}\right)$. Therefore, the ratio of the job's benefit to its cost is $\alpha\left(\frac{1}{w}\right)$. In other words, it is a function of only the wave-width. Smaller the wave-width, larger the savings in completion time per unit of cache spent. This is illustrated in Figure 2 comparing two jobs with wave-widths of 2 and 4. We show the savings in completion time when they are given 100% memory-locality. Therefore, in a cluster with multiple jobs, *average completion time is best reduced by favoring the jobs with smallest wave-widths.*

This can be easily extended to a scenario with multi-waved jobs. Every multi-waved job can be split in to many independent single-waved jobs, each with as many tasks as the wave-width. It is easy to see that the average completion time is best reduced by still picking the jobs that have the smallest wave-widths.

The paper is outlined as follows. We analyze traces from three different datacenters in §3. §4 describes our cache replacement policy and §5 describes the architecture of PACMan. We evaluate the benefits of PACMan in §6. §7 contrasts related work and we conclude in §8.

[3]Tasks are scheduled in descending order of input size.

	Facebook	Microsoft Bing	Yahoo!
Dates	Oct 2010	May-Dec* 2009	April 2011
Framework	Hadoop	Dryad	Hadoop
File System	HDFS [2]	Cosmos	HDFS [2]
Script	Hive [3]	Scope [25]	Pig [12]
Jobs	375K	200K	515K
Input Data	150PB	310PB	200PB
Cluster Size	3,500	Thousands	4,000
Memory**	48GB	N/A	16GB

* *One week in each month*
** *Memory per machine*

Table 1: **Details of MapReduce datasets analyzed from Facebook, Microsoft Bing, and Yahoo! clusters.**

3 Workloads in Production Clusters

In this section, we analyze workloads from three production clusters, each consisting of thousands of machines – Facebook's Hadoop cluster, Microsoft Bing's Dryad cluster, and Yahoo!'s Hadoop cluster. Together, they account over a million jobs processing more than 0.6EB of data. Looking at both Hadoop as well as Dryad installations – two major cluster computing frameworks – help us believe in the generality of our findings. Table 1 lists the relevant details of the traces and the clusters.

All three clusters co-locate storage and computation. The distributed file systems in these clusters store data in units of *blocks*. Computation frameworks split jobs in to multiple parallel *tasks*, each of which operate on one or more *blocks* of data. For each of the jobs we obtain task-level information: start and end times, size of the blocks the task reads (local or remote) and writes, the machine the task runs on, and the locations of its inputs. All clusters run a mix of production and user-generated queries. Production jobs are critical as their outputs feed front-end applications. User-generated queries are small and interactive where fast completions are helpful.

By understanding the characteristics of jobs and their inputs, we aim to arrive at a set of founding principles for the design of PACMan's cache management system.

3.1 Heavy-tailed Input Sizes of Jobs

Datacenter jobs exhibit a heavy-tailed distribution of input sizes. As plotted in Figure 3, workloads consist of very many small jobs and relatively few large jobs. In fact, 10% of overall data read is accounted by a disproportionate 96%, 90% and 82% of the smallest jobs in the Facebook, Bing and Yahoo! workloads. Figure 4 shows that job sizes – input sizes and number of tasks – indeed follow a power-law distribution, as the log-log plot shows a linear relationship. Also, the input size of a job and its number of tasks are correlated.

The skew in job input sizes is so pronounced that a

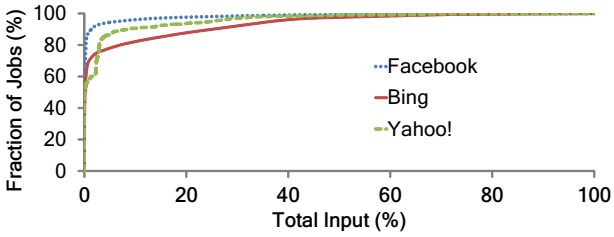
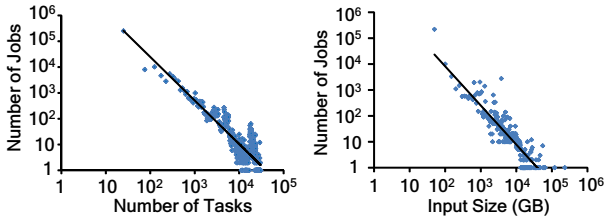


Figure 3: **Heavy-tail: Fraction of total data (Y-axis) accessed by the smallest fraction of jobs (X-axis).**



(a) Number of tasks (b) Input Size

Figure 4: **Power-law distribution of jobs (Facebook) in the number of tasks and input sizes. Power-law exponents are** 1.9 **and** 1.6 **when fitted with least squares regression.**

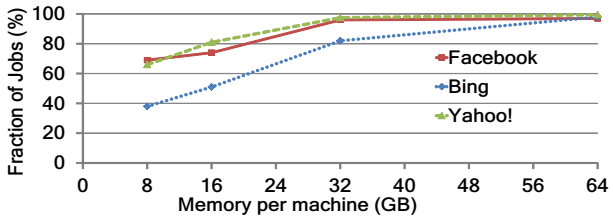


Figure 5: **Smallest fraction of active jobs (Y-axis) whose data fits in the aggregate cluster memory.**

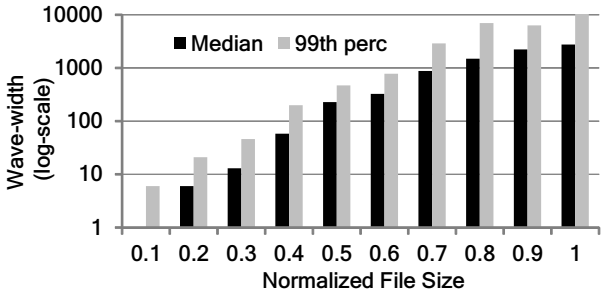


Figure 6: **Wave-width, *i.e.,* number of simultaneous tasks, of jobs as a function of sizes of files accessed. File sizes are normalized to the largest file; the largest file has size 1.**

large fraction of active jobs[4] can simultaneously fit their entire data in memory. We perform a simple simulation that looks at jobs in the order of their arrival time. The simulator assumes the memory and computation slots across all the machines in the cluster to be aggregated. It loads a job's entire input into memory when it starts and deletes it when the job completes. If the available memory is insufficient for a job's entire input, none of it is loaded. Figure 5 plots the results of the simulation. For the workloads from Facebook, Bing and Yahoo!, we see that 96%, 89% and 97% of the active jobs respectively can have their data entirely fit in memory, given an allowance of 32GB memory per server for caching.

In addition to being easier to fit a small job's input in memory, its wave-width is crucial. Recall that we can speed-up a job only when there is at least one wave-width of input cached, and in a shared cluster, average completion time is best improved by caching inputs of jobs with the smallest wave-widths. In our workloads, wave-widths correlate well with the input file size of the job. Figure 6 plots the wave-width of jobs binned by the size of their input files. Small jobs, accessing the smaller files, have significantly lower wave-width. Their wave-widths are three to four orders of magnitude smaller than the largest jobs. This is because, typically, small jobs do not have sufficient number of tasks to utilize the slots allocated by the scheduler. This implies that retaining inputs of small jobs is most likely to improve the *average completion time*. In addition, it directly improves interactive jobs where fast job completions help productivity. Our replacement policy, LIFE, rests on this motivation.

Nonetheless, large jobs are important too. Over 80% of the IO and over 90% of cluster cycles are consumed by less than 10% of the largest jobs (7% of the largest jobs in the Facebook cluster). These large jobs, in the clusters we considered, are typically revenue-generating critical production jobs feeding front-end applications. Conversations with datacenter operators indicate that speeding them up provides strategic advantages. Crucially, it also improves the overall *resource utilization* of the cluster. SIFE is our replacement policy that favors large jobs.

3.2 Data Access Patterns

Finally, we look two key traits in data access patterns – *skewed popularity* and *repeatability*.

Popularity Skew: As noted in prior work, the popularity of input data is skewed in data-intensive clusters [16]. A small fraction of the data is highly popular, while the rest is accessed sparingly. Figure 7 shows that the top 12% of popular data is accessed 10× more than the bottom third in the Bing cluster. The Facebook cluster experiences a similar skew. The top 5% of the blocks are seven times more popular than the bottom three-quarters. The skew in data access in the Yahoo! cluster falls in between the other two clusters. The upshot is that a caching scheme

[4]By active jobs we mean jobs that have at least one task running.

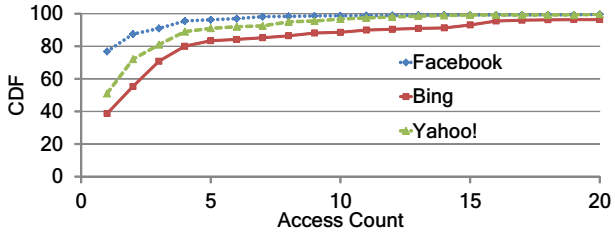


Figure 7: **Popularity of data plotted as a CDF of all blocks as a function of their access count.**

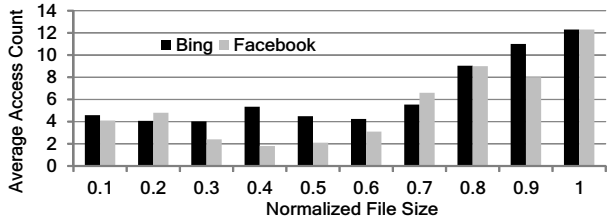


Figure 8: **Access count of files as a function of their sizes, normalized to the largest file; largest file has size 1. Large files, accessed by production jobs, have higher access count.**

should lean towards retaining the oft-accessed blocks.

Interestingly, large files have higher access counts (see Figure 8). Often they are accessed by production jobs to generate periodic (hourly) summaries, *e.g.,* financial and performance metrics, from various large logs over consolidated time intervals in the past. These intervals could be as large as weeks and months, directly leading to many of the logs in that interval being repeatedly accessed. This strengthens the idea from §3.1 that resource utilization can be improved by favoring large jobs.

Repeatability: A noteworthy statistic emerges when we look at the access counts by the corresponding jobs that read the data. We observe that *single-accessed files are spread across only* 11%, 6% *and* 7% *of jobs* in the Facebook, Bing and Yahoo! workloads. Even in these jobs, not all the data they access is singly-accessed. Repeated access of files is due to the following reasons. Iterative experimental analysis on small data increases their reuse. Large files are reused by jobs that generate consolidated summaries. Hence, we have sufficient repeatability to improve job performance by caching their inputs.

In summary, our analysis shows, (i) heavy-tailed input distribution where small jobs dominate by count, large jobs by cluster utilization, (ii) it is cheaper to speed-up small jobs due to the lower wave-width and input sizes, and (iii) reuse of data to be cached for future accesses.

4 Coordinated Cache Replacement

In this section, we discuss the rationale behind and the details of the cache replacement algorithms in PACMan.

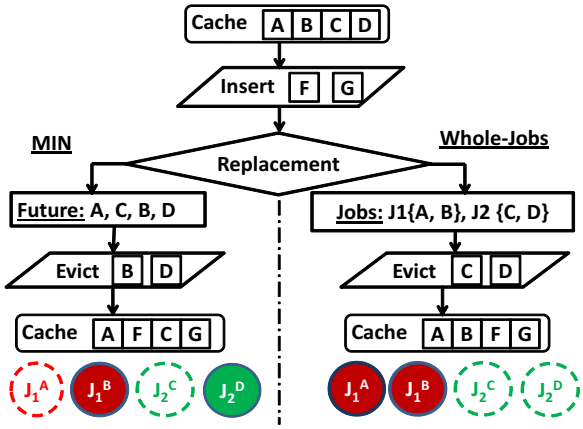


Figure 9: **Illustration of how cache hit-ratio does not necessarily improve job completion times. We consider a cache with four blocks and having to insert two new blocks. MIN evicts blocks to be accessed farthest in future. Whole-jobs preserves complete inputs of jobs.**

4.1 Whole-Jobs

We start with explaining why the traditional option of maximizing cache hit-ratio will not necessarily lead to reducing completion times of parallel jobs, describe how this influences our design, and then describe how and why we approximate wave-widths to entire jobs.

In §2, we illustrated how job completion times improve only when at least one wave-width of a job's input is cached. Therefore, for workloads consisting of parallel jobs, maximizing the cache hit-ratio might be far from optimal when it comes to improving completion times. Consider the situation depicted in Figure 9 of a 4-entry cache storing blocks A, B, C and D. Job J_1's two tasks will access blocks A and B, while job J_2's two tasks will access C and D. Both jobs consist of just a single-wave and have the all-or-nothing property where job completion time improves only if their entire input is cached.

Now, pretend a third job J_3 with input F and G is scheduled before J_1 and J_2, requiring the eviction of two blocks currently in the cache. Given the oracular knowledge that the future block access pattern will be A, C, B, then D, MIN [11] will evict the blocks accessed farthest in the future: B and D. Then, when J_1 and J_2 run, they both experience a cache miss on one of their tasks. These cache misses bound completion time for both jobs, meaning that MIN cache replacement resulted in no reduction in completion time for either J_1 or J_2.

An input-aware scheduler would recognize the dependency between each job's inputs, and instead of evicting the blocks accessed farthest-in-the-future, would evict at the granularity of a job's wave-width. In this situation with two single-wave jobs, the input-aware scheduler in Figure 9 chooses to evict the input set of J_2 (C and

D). This results in a reduction in completion time for J_1 (since its wave's entire input set of A and B is cached). J_2's completion time is unaffected. Note that the cache hit-ratio for MIN and whole-jobs is the same (50%).

Our cache replacement is built on this intuition of retaining whole wave inputs. However, wave-widths and what exact tasks run in a wave are both complicated to model. They are decided based on slot availabilities, fairness restrictions and scheduler policies. Unlike MIN, that only models the order in which requests arrive, our setting requires modeling the exact time of the request, which in turn requires modeling the speed-up due to memory-locality. The exact time of the request is important to estimate availability of compute slots in the cluster that directly dictates the wave-widths. Finally, we also have to take scheduler policies in to account. All these factors are highly variable and hard to model.

We make the problem tractable by approximating small and large wave-widths to, simply, small and large jobs (based on their input sizes). Small jobs often have very few tasks and run in one wave with a small wave-width. Modeling such waves as jobs, therefore, introduces no errors. Also, wave-widths grow proportionately with input sizes (Figure 6). This means that the larger wave-widths are for the large jobs. Therefore, we maintain the relative ordering between small and large waves despite approximating them to small and large jobs.

PACMan's cache replacement aims to retain "*whole-jobs*" inputs in the cache. The question, then, is on deciding which of the jobs' inputs to retain. Our analysis in §2 and §3 showed two things: (i) Average completion time is best improved by caching the inputs of small jobs, and (ii) A few large jobs consume most resources. The inputs of these large jobs are also accessed more often, so favoring them improves cluster utilization.

Therefore, PACMan enforces two replacement policies, one for improving small jobs (LIFE, for Largest-Incomplete-File-to-Evict) and the other for improving large jobs (SIFE, for Smallest-Incomplete-File-to-Evict). LIFE evicts blocks from the largest incompletely cached file. If all files are completely cached, it picks the largest file. SIFE deletes blocks from the smallest incompletely cached file, else the smallest complete file. The idea of picking incompletely cached files for eviction is crucial as it disturbs the fewest completely cached inputs.

4.2 LIFE and SIFE in PACMan

We now describe how LIFE and SIFE are implemented in a cluster where caches are distributed.

PACMan's architecture consists of a *coordinator* that contains a global view of all the distributed *client* caches. The coordinated infrastucture is crucial as it disturbs as few completely cached inputs as possible by evicting

```
procedure FILETOEVICT_LIFE(Client c)
    cFiles = fileSet.filter(c)              ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()   ▷ Aging
    if f == null then                       ▷ No old files to age out
        f = cFiles.getLargestIncompleteFile()
    else if f == null then                  ▷ Only complete files left
        f = cFiles.getLargestCompleteFile()
    end if
    return f.name                           ▷ File to evict
end procedure

procedure FILETOEVICT_SIFE(Client c)
    cFiles = fileSet.filter(c)              ▷ Consider only c's files
    f = cFiles.olderThan(window).oldest()   ▷ Aging
    if f == null then                       ▷ No old files to age out
        f = cFiles.getSmallestIncompleteFile()
    else if f == null then                  ▷ Only complete files left
        f = cFiles.getSmallestCompleteFile()
    end if
    return f.name                           ▷ File to evict
end procedure

procedure ADD(Client c, String name, Block bId)
    File f = fileSet.getByName(name)
    if f == null then
        f = new File(name)
        fileSet.add(f)
    end if
    f.addLocation(c, bId)                   ▷ Update properties
end procedure
```

Pseudocode 1: **Cache replacement policy – LIFE and SIFE – from the perspective of the coordinator.**

from the incomplete files. We postpone detailed description of our architecture to §5. Since LIFE and SIFE are global cache replacement policies modeled on "whole-jobs", they are implemented at the coordinator. Pseudocode 1 describes the LIFE and SIFE algorithms. In the following description, we use the terms file and input interchangeably. If all blocks of a file are cached, we call it a *complete file*; otherwise it is an *incomplete file*.

When a client runs out of cache memory it asks the coordinator for a file whose blocks it can evict, by calling FILETOEVICT() (LIFE or SIFE). To make this decision, LIFE first looks whether the client's machine caches the blocks of any incomplete file. If there are multiple such files, LIFE picks the largest one and returns it to the client. There are two points worth noting. First, by picking an incomplete file, LIFE ensures that the number of fully cached files does not decrease. Second, by picking the largest incomplete file, LIFE increases the opportunity for more small files to remain in cache. If the client does not store the blocks of any incomplete file, LIFE looks at the list of complete files whose blocks are cached by the client. Among these files, it picks the largest one.

By picking the largest one, it increases the probability of multiple small files being cached in future.

SIFE is diametrically opposite to LIFE and attempts to rid the cache of small files. To evict a block, it first checks if there are incomplete files and picks the *smallest* among them. If there are no incomplete files, it favors the large jobs by picking the smallest complete file.

To avoid cache pollution with files that are never evicted (*e.g.,* very small or large files), we also implement a window based aging mechanism. Before checking for incomplete and complete files, both LIFE and SIFE check whether the client stores any blocks of a file that has not been referred to for at least *window* time period. Among these files, it picks the one that has been accessed the least number of times. This makes it flush out the less popular blocks. Specifically, this helps in evicting singly-accessed blocks as they are unlikely to be used further. In practice, we set the window to be large (*e.g.,* hours), and it has had limited impact on most workloads.

Finally, upon caching a block, a client contacts the coordinator by calling ADD(). This allows the coordinator to maintain a global view of the system-wide in-memory cache. Similarly, when a block is evicted, the client calls REMOVE() to update the coordinator's global view. We have omitted the pseudocode for this for brevity.

PACMan operates in conjunction with the distributed file system. However, in practice, they are insulated from the job identifiers that access them. Therefore, we approximate the policy of whole-jobs to maximizing the number of whole files that are present in cache, an approximation that works well (§6.4).

5 PACMan: System Design

PACMan globally coordinates access to its caches. Without global coordination, it would be hard to ensure that a job's different input blocks distributed across machines are viewed in unison, and its tasks get memory-locality. Recall from §4 that a coordinated service was crucial for LIFE and SIFE to know about incomplete files.

Therefore, the two primary requirements from PACMan are, (*a*) support queries for the set of locations where a block is cached, and (*b*) mediate cache replacement globally across machines in the cluster.

PACMan's architecture consists of a central *coordinator* and a set of *clients* located at the storage nodes of a cluster (Figure 10). Blocks are added to the PACMan clients. PACMan clients update the coordinator when the state of the cache changes (when a block is added or removed). The coordinator uses these updates to maintain a mapping between every cached block and the locations of client nodes that cache it. Computing frameworks work with the coordinator to achieve memory-locality.

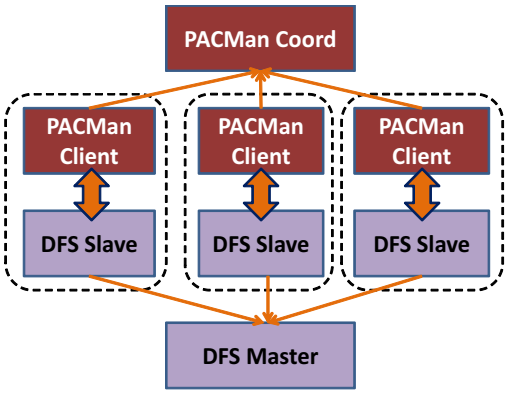


Figure 10: PACMan **architecture. The central *coordinator* controls cache management of the distributed *clients*. The dotted rectangles denote machines. Thick arrows represent data flow while thin arrows denote the metadata flow.**

The client's main job is to serve cached blocks, as well as cache new blocks. We choose to cache blocks at the *destination*, *i.e.,* the machine where the task executes as opposed to the *source*, *i.e.,* the machine where the input is stored. This allows an uncapped number of cached replicas, in turn increasing chances of memory-locality. Memory-local tasks contact the local PACMan client to check if its input data is present. If not, it fetches it from disk via a DFS slave. If the task reads data from disk, it puts it in cache of the local PACMan client and the client updates the coordinator with information about the newly cached block. Data flow is local in PACMan as remote memory access is likely constrained by the network.

Pluggable eviction policies: PACMan's architecture is agnostic to the cache replacement algorithms, enabling DFSs to plug in their own replacement policy of choice (§4). Its global cache view supports implementing coordinated replacement. We discussed LIFE and SIFE in §4, it can also be leveraged to implement policies like global LFU and global LRU. Also, since cloud DFSs typically do not support modification of existing blocks, eviction to maintain cache consistency is unnecessary.

Fault Tolerance: As the coordinator is not on the critical path, its failure does not hamper the job's execution. It can fall back to reading from disk. However, we include a secondary coordinator that functions as a cold standby. Since the secondary coordinator has no cache view when it starts, clients periodically send updates to the coordinator informing it of the state of its cache. The secondary coordinator uses these updates to construct the global cache view. Clients do not update their cache when the coordinator is down. Note that jobs are transparent to the failure and reconstruction of the coordinator.

Scalability: Nothing precludes distributing the central coordinator across different machines to avoid having it be a bottleneck. We have, however, found that the scala-

bility of our system suffices for our workloads (see §6.6).

Interface to File Systems: Now we describe the interfaces exposed by PACMan and how DFS implementations can leverage it to obtain caching functionality. PACMan is architected as a separate service. Every DFS *registers* with the coordinator requesting a certain cache size per machine. The *getBlockLocations*() call to the coordinator returns the locations of nodes where data has been cached in by the PACMan client. Memory-locality can be implemented as follows. The job scheduler usually queries the DFS for the list of locations where an input block of a task is present. On such a query, the DFS master checks with the PACMan coordinator for locations where a block might be cached in memory (*getBlockLocations*) and returns it to the scheduler, along with the disk locations. The task is then scheduled preferably on machines with memory-locality. Clients offer two simple interfaces – *getBlock*() and *putBlock*() to retrieve a cached block and cache a new block respectively.

6 Evaluation

We built PACMan and modified HDFS [2] to leverage PACMan's caching service. The prototype is evaluated on a 100-node cluster in Amazon EC2 [1] using the workloads derived from the Facebook and Bing traces (§3). To compare at a larger scale against a wider set of caching techniques, we use a trace-driven simulator that performs a detailed replay of task logs. We first describe our evaluation setup before presenting our results.

6.1 Setup

Workload: Our workloads are derived from the Facebook and Bing traces described in §3, representive of Hadoop and Dryad systems. The key goals during this derivation was to preserve the original workload's characteristics, specifically the heavy-tailed nature of job input sizes (§3.1), skewed popularity of files (§3.2), and proportional load experienced by the original clusters.

We meet these goals as follows. We replay jobs with the same inter-arrival times and input file sizes as in the original workload. This helps us mimic the load experienced by the original clusters as well as the access patterns of files. However, we scale down the file sizes proportionately to reflect the smaller size of our cluster and, consequently, reduced aggregate memory. Thereby, we ensure that there is sufficient memory for the same fraction of jobs' input as in the original workload. We confirmed by simulation (described shortly) that performance improvements with the scaled down version matched that of the full-sized cluster. Also, since we did not have access to the original scripts (Hive [3] or Scope [25]) that ran these jobs, we replaced it with canonical examples of wordcount and grep. As the focus of our work is on data reads and not the exact computation, we believe this to be an acceptable approximation. Moreover, these tasks are predominantly IO-intensive making their durations less dependent on the exact computation performed [5].

Cluster: We deploy our prototype on 100 Amazon EC2 nodes, each of them double-extra-large machines [1] with 34.2GB of memory, 13 cores and 850GB of storage. PACMan was allotted 20GB of cache per machine; we evaluate the PACMan's sensitivity to cache size in §6.5.

Trace-driven Simulator: We use a trace-driven simulator to evaluate PACMan at larger scales and longer durations. The simulator performed a detailed and faithful replay of the task-level traces of Hadoop jobs from Facebook and Dryad jobs from Bing. It preserved the read/write sizes of tasks, replica locations of input data as well as job characteristics of failures, stragglers and recomputations [7]. The simulator also mimicked fairness restrictions on the number of permissible concurrent slots, a key factor for the number of waves in the job.

We use the simulator to test PACMan's performance at the scale of the original datacenters, as well as to mimic ideal schemes like MIN and "whole-jobs" (§4.1).

Compared Schemes: Our implementation and simulator replaced blocks in cache using LIFE and SIFE, traditional cache replacement schemes like LRU and LFU as well as ideal schemes like MIN and "whole-jobs" (§4.1).

Metrics: We concentrate on evaluating PACMan's two primary aspects – scheduling tasks on machines that contain the input cached in memory and replacing cached blocks to impact job durations. The metrics we use to evaluate, hence, are hit-ratio and average job completion time. The baseline for our deployment is Hadoop-0.20.2. The trace-driven simulator compared with currently deployed versions of Hadoop and Dryad.

The following is a summary of our results.

- Average completion times improve by 53% with LIFE, cluster utilization improves by 52% with SIFE, beating competitor replacement schemes.
- Completion times of small jobs reduce by 77% with LIFE in the Facebook and Bing workloads. Large jobs improve by 44% and 50% with SIFE (§6.2).
- Considering *incompleteness* of a file using coordination is crucial in LIFE and SIFE; without it, there is a 2× increase in average completion time (§6.2).
- PACMan's memory-local task scheduling can better exploit even the local file buffer caches by 3× compared to vanilla Hadoop (§6.3).

[5] Analysis of the Facebook workload showed an average CPU utilization of 16% despite machines running tasks (average slot utilization) for 78% of the time. This shows that tasks are IO-bound.

Bin	Tasks	% of Jobs		% of Resources	
		FB	Bing	FB	Bing
1	$1-10$	85%	43%	8%	6%
2	$11-50$	4%	8%	1%	5%
3	$51-150$	8%	24%	3%	16%
4	$151-500$	2%	23%	12%	18%
5	>500	1%	2%	76%	55%

Table 2: **Job size distributions. The jobs are binned by their sizes in the scaled-down Facebook and Bing workloads.**

6.2 Reduction in Completion Time

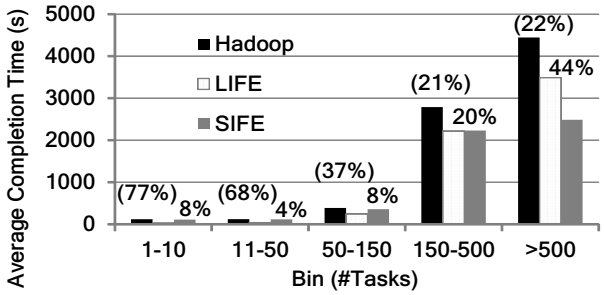
To separate the effect of PACMan's memory-locality on different jobs, we binned them by the number of map tasks they contained in the scaled-down workload. Table 2 shows the distribution of jobs by count and resources. The Facebook workload is dominated by small jobs – 85% of them have ten or fewer tasks. The Bing workload, on the other hand, has the corresponding fraction to be smaller but still sizeable at 43%. When viewed by the resources consumed, we obtain a different picture. Large jobs (bin-5) that are only 1% and 2% in the cluster consume a disproportionate 76% and 55% of all resources. The skew between small and large jobs is higher in the Facebook workload than the Bing workload.

We measure the average completion time of jobs from our deployment on EC2. Figure 11 plots the average completion times of jobs, and their relative improvement compared to Hadoop. *LIFE favors small jobs (*e.g., *bin-*1*) while SIFE improves large jobs (*e.g., *bin-*5*)*.

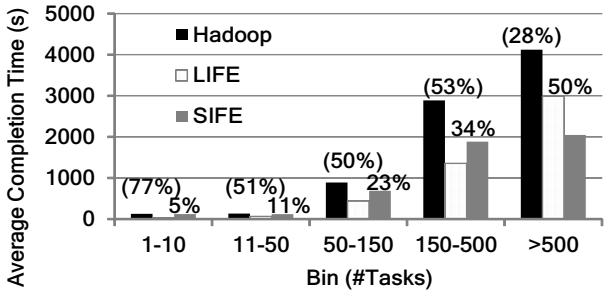
We discuss LIFE first. In both workloads, small jobs (bins 1 and 2) benefit considerably. Jobs in bin-1 see their average completion time reduce by 77% with the gains continuing to hold in bin-2. As a direct consequence of LIFE favoring small jobs, 74% of jobs in the Facebook workload and 48% of jobs in the Bing workload have *all* their tasks memory-local. Nonetheless, large jobs benefit too (bin-5) seeing a healthy improvement of 22% and 28% in the two workloads. A point to highlight is LIFE's automatic adaptability. While it favors inputs of small jobs, the benefits automatically spill over to large jobs when there is spare cache space (see bins 3 and 4).

SIFE improves the large jobs (bin-5) by 44% and 50% in the two workloads (Figure 11). Given that the large jobs consume majority of cluster resources, these translate to significant efficiency in cluster utilization. However, this comes at the expense of small jobs that see little improvement. After retaining inputs of large jobs, the remaining cache is insufficient for small jobs.

Between the two, LIFE is better for completion time averaged across all the jobs. In the Facebook and Bing workloads, LIFE improves completion times by 53% and 52%; SIFE's corresponding values are 21% and 31%. This is not surprising given the difference in cache space



(a) Facebook Workload



(b) Bing Workload

Figure 11: **Average completion times with LIFE and SIFE cache replacement, for Facebook and Bing workloads. Relative improvements compared to Hadoop are marked for each bin; LIFE improvements are in parentheses.**

required to speed up small and large jobs. However, the hit-ratios achieved by SIFE are higher than LIFE (43% and 39% vs. 58% and 62% for Facebook and Bing workloads), improving efficiency. Overall, LIFE and SIFE beat competitor replacement algorithms in average completion time and cluster utilization, respectively (§6.4).

Table 3 rigorously elaborates on the distribution of the improvements. The encouraging aspect is the tighter distribution in bin-1 with LIFE and bin-5 with SIFE, where the median and 95^{th} percentile values differ by at most 6% and 5%, respectively. Interestingly, the Facebook results in bin-2 with LIFE and the Bing results in bin-4 with SIFE are spread tighter than in the other workload.

As supporting evidence, we see that our memory-local tasks run $10.8\times$ faster than those that read data from disk.

Value of Incompleteness: An important aspect of LIFE and SIFE is its consideration of incompleteness of a file during eviction, preferring to evict blocks from already incomplete files since they are unlikely to be useful. We test its value with two schemes, LFE and SFE. LFE and SFE are simple modifications to LIFE and SIFE to not factor the incompleteness while evicting. LFE evicts the largest file in the cache, SFE evicts the smallest file in cache. Figure 12 shows the results for both workloads. Large jobs (bin-5) are hurt most. The performance of LFE is $2\times$ worse than LIFE in bin-4 and bin-5. SFE

LIFE

Bin	Tasks	Facebook			Bing		
		Mean	**Median**	**95^{th} perc.**	**Mean**	**Median**	**95^{th} perc.**
1	$1-10$	28s (77%)	27s (77%)	33s (74%)	29s (77%)	35s (75%)	44s (69%)
2	$11-50$	39s (68%)	43s (70%)	44s (70%)	65s (51%)	81s (42%)	104s (31%)
3	$51-150$	246s (37%)	252s (42%)	348s (36%)	443s (50%)	466s (54%)	502s (54%)
4	$151-500$	2901s (21%)	2302s (21%)	2801s (24%)	1357s (53%)	1502s (50%)	1709s (45%)
5	>500	3489s (22%)	3557s (18%)	4777s (5%)	2989s (28%)	2772s (37%)	2902s (37%)

SIFE

Bin	Tasks	Facebook			Bing		
		Mean	**Median**	**95^{th} perc.**	**Mean**	**Median**	**95^{th} perc.**
1	$1-10$	111s (8%)	117s (-2%)	135s (-5%)	121s (5%)	133s (5%)	142s (1%)
2	$11-50$	116s (4%)	118s (16%)	141s (4%)	119s (11%)	141s (-1%)	147s (3%)
3	$51-150$	358s (8%)	381s (12%)	414s (24%)	689s (23%)	703s (30%)	891s (19%)
4	$151-500$	2231s (20%)	2009s (31%)	2809s (23%)	1884s (34%)	1909s (37%)	2109s (32%)
5	>500	2487s (44%)	2201s (49%)	2581s (48%)	2047s (50%)	2294s (48%)	2509s (46%)

Table 3: **EC2 Results for Facebook and Bing workloads. For LIFE and SIFE strategies, we present the mean, median and 95^{th} percentile job completion times. The numbers in parentheses represent the reduction compared to baseline Hadoop.**

	Scale	Cache	LIFE-bin1		SIFE-bin5	
			FB	**Bing**	**FB**	**Bing**
EC2	100	20GB	77%	77%	44%	50%
Simulator	$1000's*$	32GB	78%	74%	49%	51%

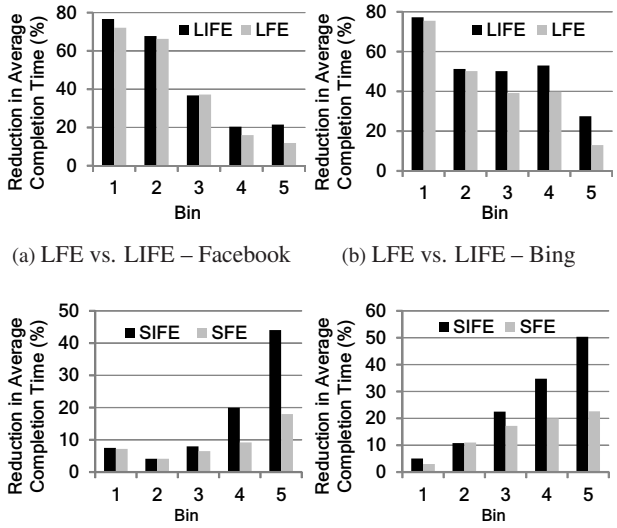
** Original cluster size*

Table 4: **Summary of results from deployments on EC2 and the trace-driven simulator.**

has a nearly $2.5\times$ difference compared to SIFE in bin-4 and bin-5, that it is designed to favor. This strongly underlines the value of coordinated replacement in PACMan by looking at global view of the cache. Interestingly, jobs in bin-1 are unaffected. While LIFE and LFE differ in the way they evict blocks of the large files, there are enough of them to leave inputs of small jobs untouched. SIFE never improved small jobs and SFE has a similar behavior of not retaining their inputs. However, do note the $\sim$20% drop in bin-3 with SFE compared to SIFE. The drop in performance is despite comparable hit-ratios. LFE's hit-ratios are 39% and 40% in the Facebook and Bing workloads, SFE's hit-ratios are 53% and 57%.

Simulation: We use the trace-driven simulator to assess its performance on a larger scale of thousands of machines (same size as in the original clusters). The simulator assumes 32GB of memory per machine for PACMan's cache. LIFE improves the average completion times of small jobs (bin-1) by 78% and 74% for the Facebook and Bing workloads. SIFE's results too are comparable to our EC2 deployment with jobs in bin-5 improving by 49% and 51% in the two workloads. This increases our confidence in the large-scale performance of PACMan as well as our methodology for scaling down the workload. Table 4 shows a comparative summary of the results.

We now proceed to dissect and better understand PACMan's benefits due to its two primary aspects – memory-local scheduling of tasks (§6.3) and cache replacement schemes targeting job improvements (§6.4).



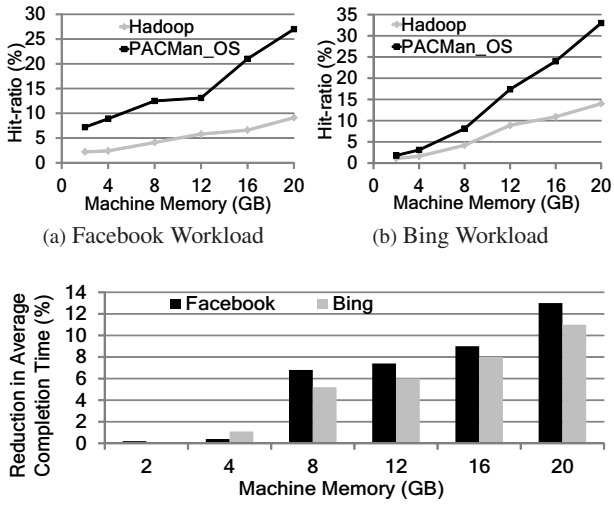
(a) LFE vs. LIFE – Facebook (b) LFE vs. LIFE – Bing

(c) SFE vs. LIFE – Facebook (d) SFE vs. SIFE – Bing

Figure 12: **Value of using incompleteness. LFE evicts the largest file in cache, SFE evicts the smallest in the cache.**

6.3 Memory-Local Task Scheduling

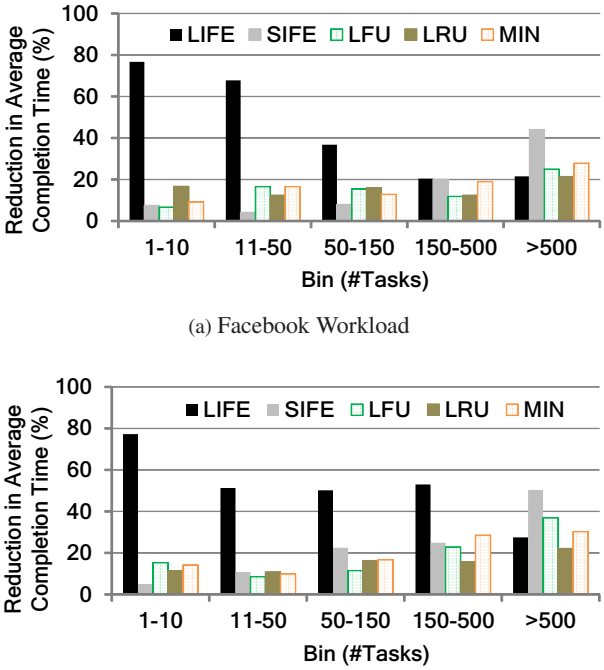
One of PACMan's features is to schedule tasks on machines that have its input cached in memory. We analyze the value of such scheduling by isolating its benefits using a simplified implementation,PACMan_OS, that only leverages the unmodified local file buffer caches.



(a) Facebook Workload (b) Bing Workload

(c) Job Improvement

Figure 13: **Using the local file buffer cache. We compare the hit-ratios ((a) and (b)) and reduction in average completion time ((c)), between PACMan_OS and vanilla Hadoop.**



(a) Facebook Workload

(b) Bing Workload

Figure 14: **Comparison between LIFE, SIFE, LFU, LRU and MIN cache replacements.**

We turn off the PACMan coordinator and all the clients. The job scheduler in Hadoop itself maintains an estimate of the machines where each data block is cached in the local file buffer caches. After scheduling each task, the job scheduler updates the corresponding input block's cached location to the machine that the task read its data off. Subsequent scheduling of tasks that read the same input is preferably done for memory-locality [6].

Of interest to us is the buffer cache hit-ratio, *i.e., fraction of data that is read from the buffer cache*. We compare PACMan_OS with vanilla Hadoop. Hadoop does not explicitly schedule tasks for memory-locality. We verify whether a task read its data from cache or disk using iostat [4] values recorded at the start and end of a task.

Figures 13a and 13b present hit-ratios for varying memory sizes per machine (2GB to 20GB), indirectly controlling the buffer cache size. Note the low slope of the curve for stock Hadoop – since it is agnostic to cache locations, it does not capitalize on more blocks being cached as memory sizes increase. In both workloads, we see moderate difference in hit-ratios for small cache sizes, while the two curves have a large divergence for higher memory sizes. With limited memory, data is less likely to be retained in cache, so Hadoop's policy of scheduling tasks without being aware of their inputs' cached locations does not cause much harm. However, with larger memory sizes, where more blocks are likely to be retained in cache, PACMan_OS's targeted scheduling stands out with a $2.2\times$ and $3\times$ difference in the two workloads. A subtle point to note is that since blocks are cached at the source, vanilla Hadoop's performance would be similar to PACMan_OS once all the three replicas of a block are in their respective buffer caches. However, since majority of blocks have an access count of ≤ 4, we do not have the luxury of cache misses (§3.2).

Notwithstanding the improved hit-ratio, we do not observe much improvement in completion times (Figure 13c). This supports our assertion that improving hit-ratios do not necessarily accelerate parallel jobs.

In summary, we demonstrate that PACMan's awareness of cached locations of blocks achieves better hit-ratios even using the local buffer caches but limited job improvements. We now proceed to understand the benefits of PACMan's coordinated cache replacement.

6.4 LIFE and SIFE: Cache Replacement

In addition to our coordinated eviction algorithms, LIFE and SIFE, our prototype also implements traditional cache replacement techniques like LRU and LFU. Figure 14 compares their performance. Between LRU and LFU, the latter turns out to be marginally better for small jobs and significantly more beneficial for large jobs. LIFE outperforms both LRU and LFU for small jobs while achieving comparable performance for large jobs. SIFE invests all its cache space for jobs in large jobs and hence lags behind LRU and LFU for jobs in bin-1.

[6] Blocks are assumed to expire off the cache after a time period; 30 minutes gave the maximum hit-ratio in our experiments.

Scheme	Facebook		Bing	
	% Job Saving	Hit Ratio (%)	% Job Saving	Hit Ratio (%)
MIN	13%	63%	30%	68%
LRU	15%	36%	16%	34%
LFU	10%	47%	21%	48%
LIFE	53%	43%	52%	39%
SIFE	21%	58%	31%	62%

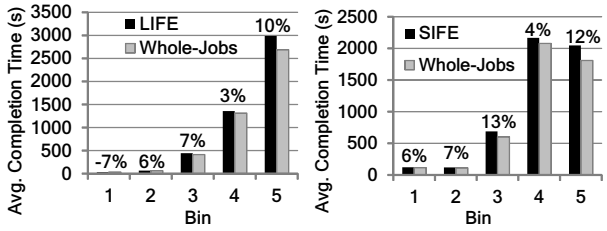
Table 5: **Performance of cache replacement schemes. LIFE is best in improving average completion times. SIFE is second only to MIN in achieving hit-ratios.**

In fact, LIFE and SIFE outperform even the cache hit-ratio optimal eviction algorithm, MIN [11]. MIN deletes the block that is to be accessed farthest in the future. As Figure 14 shows, not taking the wave-width constraint of jobs into account hurts MIN. Since it is agnostic to the set of blocks that are accessed in parallel, small all-or-nothing jobs are especially hurt. LIFE is $7.1\times$ better than MIN in bin-1. Further, note the $2.5\times$ between LIFE and MIN in bin-3 and bin-4 in the Bing workload. This is due to LIFE's property where large jobs automatically benefit once small jobs have been satisfied. While MIN's performance for bin-5 is comparable to LIFE, SIFE outperforms MIN by a factor of $1.7\times$ in this bin. Overall, this is despite a lower cache hit-ratio (58% for MIN versus 46% for LIFE and 54% for SIFE). This underscores the key principle and differentiator in LIFE and SIFE – coordinated cache replacement to minimize job completion time as opposed to simply focusing on hit-ratios.

LIFE beats all other replacement algorithms when we measure reduction in average completion time, across all the jobs. SIFE's hit-ratio, likewise, is next only to MIN which has perfect future knowledge. For speeding up jobs, SIFE is second only to LIFE. Table 5 lists reduction in average completion time and hit-ratios of all the schemes. LIFE wins on average completion time because small jobs provide the highest improvement-to-space ratio. SIFE is second only to MIN in hit-ratio because large inputs are accessed more often.

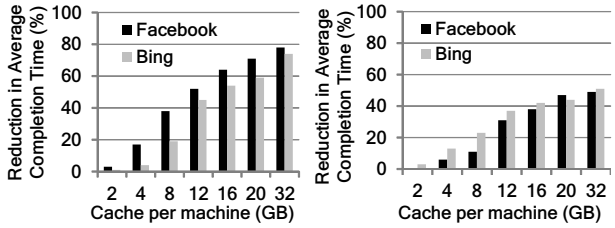
Whole-jobs: Recall from §4.2 that our desired policy is to retain the input of as many *whole-jobs* as possible. As job-level information is typically not available at the file system or caching level, for ease and cleanliness of implementation, LIFE and SIFE approximates this by retaining as many whole *files* as possible. We estimate the error in our approximation using our simulator.

LIFE and SIFE are on par with the eviction policy of whole-jobs for small jobs (Figure 15). This is because small jobs typically tend to operate on single files, therefore the approximation does not introduce errors. In fact, in the Bing trace, 67% of jobs operate on only one file. For larger jobs (bin-5), that access multiple files, the policy of whole-jobs proves to be better than LIFE and SIFE by 10% and 12% respectively. The comparable average



(a) LIFE (b) SIFE

Figure 15: **Comparing LIFE and SIFE with the the policy of *whole-jobs*, for the Bing workload. The differences in average completion time are marked for each bin.**



(a) LIFE (bin-1) (b) SIFE (bin-5)

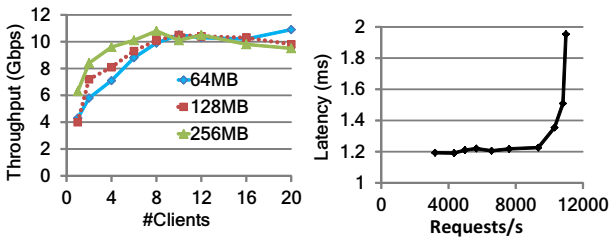
Figure 16: **LIFE's and SIFE's sensitivity to cache size.**

completion times makes us conclude that the approximation is a reasonable trade-off for the significant implementation ease but we do believe there is room for improvement. Extending our cache replacement to infer clusters of files accessed together is part of future work.

6.5 Cache Size

We now evaluate PACMan's sensitivity to available cache size. Figure 16 plots the reduction in completion times with LIFE and SIFE as the budgeted cache space at each PACMan client varies (between 2GB and 32GB). We focus on bin-1 for LIFE and bin-5 for SIFE. The encouraging observation is that both LIFE and SIFE react gracefully to reduction in cache space. As the cache size reduces from 20GB on to 12GB, the performance of LIFE and SIFE under both workloads hold to provide appreciable reduction of at least 31% and 37%, respectively.

For lower cache sizes ($\leq$ 8GB), the workloads have a stronger influence on performance. While both workloads have a strong heavy-tailed distribution, recall from Table 2 that the skew between the small jobs and large jobs – by count as well as resources – is higher in the Facebook workload. The high fraction of small jobs in the Facebook workload ensures that LIFE's performance drops much more slowly. Even at lower cache sizes, there are sufficient small jobs whose inputs can be retained by LIFE. Contrast with the sharp drop for caches



(a) PACMan Client (b) PACMan Coordinator

Figure 17: PACMan **Scalability. (a) Number of simultaneous cache reads that can be serviced by the** PACMan **client, (b) Number of simultaneous client updates that can be serviced by the** PACMan **coordinator.**

sizes $\leq$ 8GB for the Bing workload.

SIFE exhibits the opposite behavior. Large jobs show moderate but not steep improvement with increasing hit-ratio. Facebook workload's heavy skew in input sizes of large jobs means that there is not enough memory to provide sizeable benefits for them, despite favoring them. On the Bing workload, with lesser skew, SIFE reacts better. Note that even as the cache size drops to 8GB SIFE results in 23% reduction in job completion times.

Our sensitivity analysis on cache size shows that PACMan is resilient to shrinking of available space. For very low cache sizes, workload characteristics results in interesting behaviors from LIFE and SIFE.

6.6 Scalability

We now probe the scalability limits of the PACMan coordinator and client. The client's main functionality is to provide and cache blocks for tasks. The coordinator communicates with multiple clients to update block locations and coordinate LIFE and SIFE. Therefore, we measure the throughput when communicating with the client and latency when dealing with the coordinator.

PACMan **Client:** We stress the PACMan client to understand the number of simultaneous tasks it can serve before its capacity saturates. Each of the tasks read a block from the client's cache. Figure 17a reports the aggregate throughput for block sizes of 64MB, 128MB and 256MB. For block sizes of 128MB, we see that the client saturates at 10 tasks. Increasing the number of clients beyond this point results in no increase in aggregate throughput. Halving the block size to 64MB only slightly nudges the saturation point to 12 tasks. We believe this is due to the overheads associated with connection management. Connections with 256MB blocks peak at 8 tasks beyond which the throughput stays constant. The set of block sizes we have tested represent the commonly used settings in many Hadoop installations. Also, since Hadoop installations rarely execute more than 8 or 10 map tasks per machine, we conclude that our client scales sufficiently to handle the expected load.

PACMan **Coordinator:** Our objective is to understand the latency added to the task because of the PACMan client's communication with the PACMan coordinator. Since we assume a single centralized coordinator, it is crucial that it supports the expected number of client requests (block updates and LIFE eviction). We vary the number of client requests directed at the server per second and observe the average latency to service those requests. As Figure 17b shows, the latency experienced by the requests stays constant at $\sim$1.2ms till $10,300$ requests per second. At this point, the coordinator's processing overhead starts increasing the latency. The latency nearly doubles at around $11,000$ requests per second. Recently reported research on framework managers [10] show that the number of requests handled by the centralized job managers of Hadoop is significantly less ($3,200$ requests/second). Since a task makes a single request to the coordinator via the client, we believe the coordinator scales well to handle expected loads.

7 Related Work

There has been a humbling amount of work on in-memory storage and caches. While our work borrows and builds up on ideas from prior work, the key differences arise from (i) operating in a datacenter setting with parallel jobs, thereby leading to a coordinated system, and (ii) focusing on job speed up as opposed to hit-ratio.

RAMCloud [20] and prior work on databases such as MMDB [17] propose storing all data in RAM. While this is suited for web servers, it is unlikely to work in data-intensive clusters due to capacity reasons – Facebook has 600$\times$ more storage on disk than aggregate memory. Our work thus treats memory as a constrained cache, focusing on how to best use it to speed up jobs.

Global memory caching systems such as the GMS project [5], NOW project [6] and others [15] have coordinated caching among nodes by using the memory of a remote machine instead of spilling to disk. PACMan's caches only serves tasks on the local node, designed based on the vast difference between local memory and network throughputs. However, nothing in the design precludes adding the notion of a global memory view. Crucially, PACMan's coordinated caching considers job access patterns for replacement.

Web caches have identified the difference between byte hit-ratios and request hit-ratios, i.e., the value of having an entire file cached to satisfy a request [9, 13, 24]. Request hit-ratios are best optimized by retaining small files [27], a notion we borrow. We build up on it by addressing the added challenges in data-intensive clusters. Files in our setting are distributed across ma-

chines unlike web caches necessitating coordinated replacement. Web caches do not identify benefits for partial cache hits for files, whereas large jobs in datacenters do see benefits with partial memory-locality. This leads to more sophistication in our cache replacement like carefully evicting parts of an incomplete file. The analogy with web caches would not be a web request but a web *page* - collection of multiple web objects (.gif, .html). Web caches, to the best of our knowledge, have not considered cache replacment to optimize at that level.

Distributed filesystems such as Zebra [18] and xFS [8] developed for the Sprite operating system [23] make use of client-side in-memory block caching, also suggesting using the cache only for small files. However, these systems make use of relatively simple eviction policies and do not coordinate scheduling with locality since they were designed for usage by a network of workstations.

Cluster computing frameworks such as Piccolo [26] and Spark [22] are optimized for iterative machine learning workloads. They cache data in memory after the first iteration, speeding up further iterations. The key difference with PACMan is that since we assume no application semantics, our cache can benefit multiple and a greater variety of jobs. We operate at the storage level and can serve as a substrate for such frameworks to build upon.

8 Conclusion

We have described PACMan, an in-memory caching system for data-intensive parallel jobs. Parallel jobs run multiple tasks simultaneously and a job is sped up only when inputs of all such parallel tasks are cached. By globally coordinating access to the distributed local caches, PACMan ensures that a job's different tasks distributed across machines all get their data from cache. PACMan is cognizant of the heavy-tailed distribution of job sizes in these workloads. They have very many small jobs that are user-generated and interactive, and a few large production jobs that consume most resources. PACMan's two cache replacement policies – LIFE and SIFE – are designed to favor small and large jobs, respectively. We have evaluated PACMan using a deployment on EC2, along with extensive trace-driven simulations. PACMan reduces job completion times by 53% and improves utilization by 52%. LIFE and SIFE beat their competitors in both average completion time and cluster utilization.

References

[1] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/instance-types/`.

[2] Hadoop distributed file system. `http://hadoop.apache.org/hdfs`.

[3] Hive. `http://wiki.apache.org/hadoop/Hive`.

[4] iostat - linux users manual. `http://linuxcommand.org/man_pages/iostat1.html`.

[5] The Global Memory System (GMS) Project. `http://www.cs.washington.edu/homes/levy/gms/`.

[6] The NOW Project. `http://now.cs.berkeley.edu/`.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX OSDI*, 2010.

[8] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *ACM SOSP*, 1995.

[9] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating Content Management Techniques for Web Proxy Caches. In *WISP*, 1999.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, 2011.

[11] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 1966.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *ACM SIGMOD*, 2008.

[13] L. Cherkasova and G. Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *HPCN EUrope*, 2001.

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI*, 2004.

[15] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server database architectures. In *VLDB*, 1992.

[16] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *EuroSys*, 2011.

[17] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. In *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[18] John H. Hartman and John K. Ousterhout. The zebra striped network file system, 1993.

[19] J. Ousterhout *et al.* The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In *SIGOPS Operating Systems Review*, 2009.

[20] J. Ousterhout *et al.* The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In *SIGOPS Operating Systems Review*, 2009.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[23] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM TOCS*, Feb 1988.

[24] P.Cao and S.Irani. Cost Aware WWW Proxy Caching Algorithms. In *USENIX USITS*, 1997.

[25] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[26] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX OSDI*, 2010.

[27] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *ACM SIGCOMM*, 1996.