PACMan: Coordinated Memory Caching for Parallel Jobs

Ganesh Ananthanarayanan [1], Ali Ghodsi [1,4], 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* [4] *KTH/Sweden*
{ganesha,alig,awang,shenker,istoica}@cs.berkeley.edu, dhruba@facebook.com, srikanth@microsoft.com

Abstract– Data-intensive analytics on large clusters is important for modern Internet services. As machines in these clusters have large memories, in-memory caching of inputs is an effective way to speed up these analytics 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. Indeed, a single task whose input is not cached can slow down the entire job. To meet this "all-or-nothing" property, we have built PACMan, a caching service that coordinates access to the distributed caches. This coordination is essential to improve job completion times and cluster efficiency. To this end, we have implemented two cache replacement policies on top of PACMan's coordinated infrastructure – LIFE that minimizes average completion time by evicting large incomplete inputs, and LFU-F that maximizes cluster efficiency by evicting less frequently accessed inputs. Evaluations on production workloads from Facebook and Microsoft Bing show that PACMan reduces average completion time of jobs by 53% and 51% (small interactive jobs improve by 77%), and improves efficiency of the cluster by 47% and 54%, respectively.

1 Introduction

Cluster computing has become a major platform, powering both large Internet services and a growing number of scientific applications. Data-intensive frameworks (*e.g.,* MapReduce [13] and Dryad [20]) allow users to perform data mining and sophisticated analytics, automatically scaling up to thousands of machines.

Machines in these clusters have large memory capacities, which are often underutilized; the median and 95th percentile memory 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 data-intensive jobs by caching their input data.

Data-intensive jobs, typically, have a phase where they process the input data (*e.g., map* in MapReduce [13], *extract* in Dryad [20]). This phase simply reads the raw input and writes out parsed output to be consumed during further computations. Naturally, this phase is IO-intensive. Workloads from Facebook and Microsoft Bing 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. 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, 14, 16, 18]. 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 run in parallel.[1] As and when tasks finish and vacate slots, new tasks get scheduled on them. We define the number of parallel tasks as the *wave-width* of the job.

The wave-based execution implies that small single-waved jobs have an *all-or-nothing* property – unless all the tasks get memory locality, there is no improvement in 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 the exact set of tasks that run in a wave is not of concern, we only care about the wave-width, *i.e.,* how many of them run simultaneously.

Our position is that *coordinated management* of the distributed caches is required to ensure that enough tasks of a parallel job have memory locality to improve their

[1] We use the terms "small" and "large" jobs to to refer to their input size and/or numbers of tasks.

completion time. Coordination provides a global view that can 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 coordination 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 a nutshell, LIFE calculates the wave-width of every job and favors input files of jobs with small waves, *i.e.,* lower wave-widths. It replaces cached blocks of the incomplete file with the largest wave-width. The design of LIFE is driven by two observations. First, a small wave requires caching less data than a large wave 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. Second, we need to retain the entire input of a wave to decrease the completion time. 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 average completion time, as it ignores the wave-width constraint of parallel jobs. For instance, consider a simple workload consisting of 10 equal-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 a second eviction policy, LFU-F, which aims to *maximize the efficiency of the cluster*. Cluster efficiency is defined as finishing the jobs by using the least amount of resources. LFU-F favors popular files and evicts blocks from the least accessed files. Efficiency improves every time data is accessed from cache. So files that are accessed more frequently contribute more to cluster efficiency than files that will be accessed fewer number of times.

A subtle aspect is that the all-or-nothing property is important even for cluster efficiency. This is because tasks of subsequent phases often overlap with the IO-intensive phase. For example, in MapReduce jobs, reduce tasks begin after a certain fraction of map tasks finish. The reduce tasks start reading the output of completed map tasks. Hence a delay in the completion of a few map tasks, when their data is not cached, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots effectively hurting efficiency.

We have integrated PACMan with Hadoop HDFS [2]. PACMan is evaluated by running workloads from datacenters at Facebook and Bing on an EC2 cluster. We show that overall job completion times reduce by up to 53% with LIFE and cluster efficiency improves by up to 54% with LFU-F. Notably, completion times of small jobs reduce by 77% with LIFE. LIFE and LFU-F outperform traditional schemes like LRU, LFU, and even MIN [11], which is provably optimal for hit-ratios.

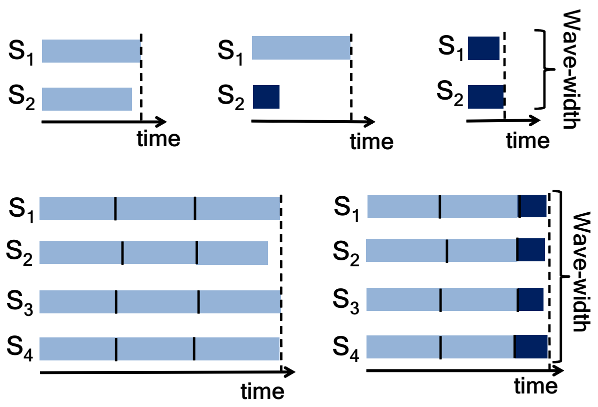


Figure 1: **Example of a single-wave (2 tasks, simultaneously) and multi-wave job (12 tasks, 4 at a time). S_i's are slots. Memory local tasks are dark blocks. Completion time (dotted line) reduces when a wave-width of input is cached.**

2 Cache Replacement for Parallel Jobs

In this section, we first explain how the concept of wave-width is important for parallel jobs, and argue that maximizing cache hit-ratio neither minimizes the average completion time of parallel jobs nor maximizes efficiency of a cluster executing parallel jobs. From first principles, we then derive the ideas behind LIFE and LFU-F cache replacement schemes.

2.1 All-or-Nothing Property

Achieving memory locality for a task will shorten its completion time. But this need not speed up the job. Jobs speed up when an entire wave-width of input is cached (Figure 1). The 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 to reduce their completion time. We refer to this as the *all-or-nothing* property. Jobs consisting of many waves improve as we incrementally cache inputs in multiples of their wave-width. In Figure 1, the single-waved job runs both its tasks simultaneously and will speed up only if the inputs of both tasks are cached. The multi-waved job, on the other hand, consists of 12 tasks and can run 4 of them at a time. Its completion time improves

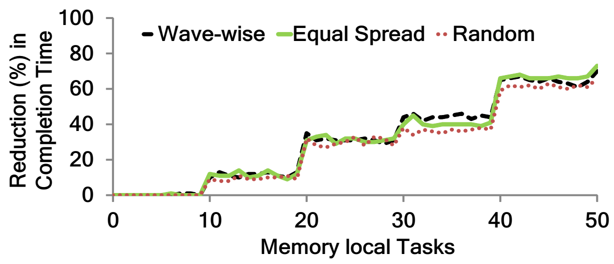


Figure 2: **Reduction in completion time of a job with** 50 **tasks running on** 10 **slots. The job speeds up, in steps, when its number of memory local tasks crosses multiples of the wave-width (*i.e.,* 10), regardless of how they are scheduled.**

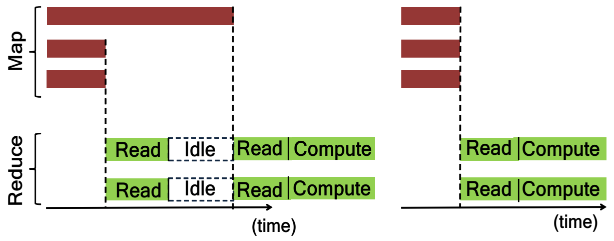


Figure 3: **All-or-nothing property matters for efficiency. In this example of a job with** 3 **map tasks and** 2 **reduce tasks, even if one map task is delayed (due to lack of memory locality), reduce tasks idle and hurt efficiency.**

in steps when any 4, 8 and 12 tasks run memory locally.

We confirmed the hypothesis of wave-widths by executing a sample job on a cluster with 10 slots (see Figure 2). The job operated on 3GB of input and consisted of 50 tasks each working on 60MB of data. Our experiment varied the number of memory-local tasks of the job and measured the reduction in completion time. The baseline was the job running with no caching. Memory local tasks were spread uniformly among the waves ("Equal Spread"). We observed the job speeding up when its number of memory-local tasks crossed 10, 20 and so forth, *i.e.,* multiples of its wave-width, thereby verifying the hypothesis. Further, we tried two other scheduling strategies. "Wave-wise" scheduled the non-memory-local tasks before memory local tasks, *i.e.,* memory local tasks ran simultaneously, and "Random" scheduled the memory local tasks in an arbitrary order. We see that the speed-up in steps of wave-width holds in both cases, albeit with slightly reduced gains for "Random". Surprisingly, the wave-width property holds even when memory local tasks are randomly scheduled because a task is allotted a slot only when slots become vacant, not a priori. This automatically balances memory local tasks and non-memory-local tasks across the compute slots.

That achieving memory locality will lower resource usage is obvious – tasks whose inputs are available in memory run faster and occupy the cluster for fewer hours. A subtler point is that the all-or-nothing constraint can also be important for cluster efficiency. This is because some of the schedulers in parallel frameworks (*e.g.,* Hadoop and MPI) allow tasks of subsequent stages to begin even before all tasks in the earlier stages finish. Such "pipelining" can hide away some data transfer latency, for example, when reduce tasks start running even before the last task in the map stage completes [3]. However, this means that a delay in the completion of some map tasks, perhaps due to lack of memory locality, results in all the reduce tasks waiting. These waiting reduce tasks waste computation slots and adversely affect efficiency. Figure 3 illustrates this overlap with an example job of three map tasks and two reduce tasks.

In summary, meeting the all-or-nothing constraint improves completion time and efficiency of parallel jobs.

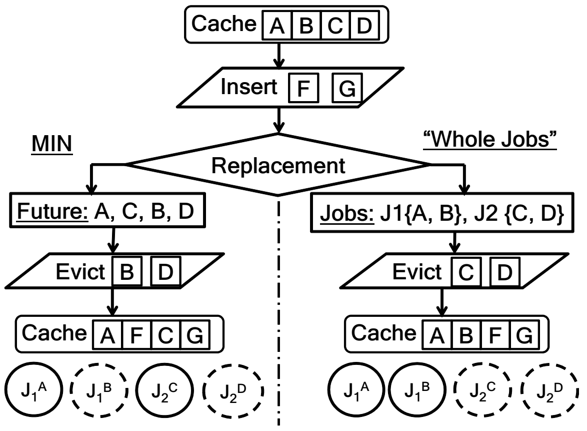


Figure 4: **Cache hit-ratio does not necessarily improve job completion. We consider a cache that has to replace two out of its four blocks. MIN evicts blocks to be accessed farthest in future. "Whole jobs" preserves complete inputs of jobs.**

2.2 Sticky Policy

Traditional cache replacement schemes that maximize cache hit-ratio do not consider the wave-width constraint of all-or-nothing parallel jobs. Consider the situation depicted in Figure 4 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 hence their job completion time improves only if their entire input is cached.

Now, pretend a third job J_3 with inputs 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 execute, they both experience a cache miss on one of their tasks. These cache misses bound their completion times, meaning that MIN cache replacement resulted in no reduction in completion time for either J_1 or J_2. Consider an alternate replacement scheme that 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 entire input set of A and B is cached). J_2's completion time is unaffected. Note that the cache hit-ratio remains the same as for MIN (50%).

Further, maximizing hit-ratio does not maximize efficiency of the cluster. In the same example as in Figure 4, let us add a reduce task to each job. Both J_1 and J_2 have two map tasks and one reduce task. Let the reduce task start after 5% of the map tasks have completed (as in Hadoop [3]). We now compare the resource consumption of the two jobs with MIN and "whole jobs" which evicts inputs of J_2. With MIN, the total resource consumption is $2(1+\mu)m + 2(0.95)m$, where m is the duration of a non-memory-local task and μ reflects the speed-up when its input is cached; we have omitted the computation of the reduce task. The policy of "whole jobs", on the other hand, expends $2(1+\mu)m + (0.95\mu + 0.05)m$ resources. As long as memory locality produces a speed-up, *i.e.,* $\mu \leq 1$, MIN consumes more resources.

The above example, in addition to illustrating that cache hit-ratios are insufficient for both speeding up jobs and improving cluster efficiency, also highlights the importance of retaining *complete* sets of inputs. Improving completion time requires retaining complete wave-widths of inputs, while improving efficiency requires retaining complete inputs of jobs. Note that retaining the complete inputs of jobs automatically meets the wave-width constraint to reduce completion times. Therefore, instead of evicting the blocks accessed farthest in the future, replacement schemes for parallel jobs should recognize the commonality between inputs of the same job and evict at the granularity of a job's input.

This intuition gives rise to the *sticky* policy. *The sticky policy preferentially evicts blocks of incomplete files.* If there is an incomplete file in cache, it sticks to its blocks for eviction until the file is completely removed from cache. The sticky policy is crucial as it disturbs the fewest completely cached inputs and evicts the incomplete files which are not beneficial for jobs.[2]

Given the sticky policy to achieve the all-or-nothing requirement, we now address the question of which inputs to retain in cache such that we minimize average completion time of jobs and maximize cluster efficiency.

2.3 Average Completion Time – LIFE

We show that in a cluster with multiple jobs, favoring jobs with the smallest wave-widths minimizes the average completion time of jobs. Assume that all jobs in the cluster are single-waved. Every job j has a wave-width of w and an input size of I. Let us assume the input of a job 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. As before, memory locality reduces its duration by a factor of μ. The factor μ is dictated by the difference between memory and disk bandwidths, but limited by additional overheads such as deserialization and decompression of the data after reading it.

To speed up a single-waved job, we need I units of cache space. On spending I units of cache space, tasks would complete in $\mu\left(\frac{I}{w}\right)$ time. Therefore the saving in completion time would be $(1-\mu)\left(\frac{I}{w}\right)$. Counting this savings for every access of the file, it becomes $f(1-\mu)\left(\frac{I}{w}\right)$, where f is the frequency of access of the file. Therefore, the ratio of the job's benefit to its cost is $f(1-\mu)\left(\frac{1}{w}\right)$. In other words, it is directly proportional to the frequency and inversely proportional to the wave-width. The smaller the wave-width, the larger the savings in completion time per unit of cache spent. This is illustrated in Figure 5 comparing two jobs with the same input size (and of the same frequency), but wave-widths of 2 and 4. Clearly, it is better to use I units of cache space to store the input of the job with a wave-width of two. This is because its work per task is higher and so the savings are proportionately more. Note that even if the two inputs are unequal (say, I_1 and I_2, and $I_1 > I_2$), caching the input of the job with lower wave-width (I_1) is preferred despite its larger input size. Therefore, in a cluster with multiple jobs, *average completion time is best reduced by favoring the jobs with smallest wave-widths* (LIFE).

This can be easily extended to a multi-waved jobs. Let the job have n waves, c of which have their inputs cached. This uses $cw\left(\frac{I}{nw}\right)$ of cache space. The benefit in completion time is $f(1-\mu)c\left(\frac{I}{nw}\right)$. The ratio of the job's benefit to its cost is $f(1-\mu)\left(\frac{1}{w}\right)$, hence best reduced by still picking the jobs that have the smallest wave-widths.

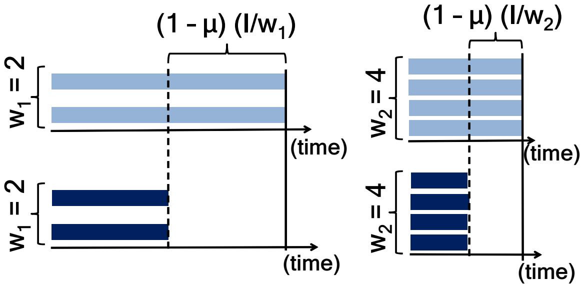


Figure 5: **Gains in completion time due to caching decreases as wave-width increases. Solid and dotted lines show completion times without and with caching (for two jobs with input of I but wave-widths of 2 and 4). Memory local tasks are dark blocks, sped up by a factor of μ.**

[2]When there are strict barriers between phases, the sticky policy does not improve efficiency. Nonetheless, such barriers are akin to "application-level stickiness" and hence stickiness at the caching layer beneath, expectedly, does not add value.

	Facebook	Microsoft Bing
Dates	Oct 2010	May-Dec* 2009
Framework	Hadoop	Dryad
File System	HDFS [2]	Cosmos
Script	Hive [4]	Scope [24]
Jobs	375K	200K
Input Data	150PB	310PB
Cluster Size	3,500	Thousands
Memory per machine	48GB	N/A

* *One week in each month*

Table 1: **Details of Hadoop and Dryad datasets analyzed from Facebook and Microsoft Bing clusters, respectively.**

2.4 Cluster Efficiency – LFU-F

In this section, we derive that retaining frequently accessed files maximizes efficiency of the cluster. We use the same model for the cluster and its jobs as before. The cluster consists of single-waved jobs, and each job j has wave-width w and input size I. Duration of tasks are proportional to their input sizes, $\left(\frac{I}{w}\right)$, and achieving memory locality reduces its duration by a factor of μ.

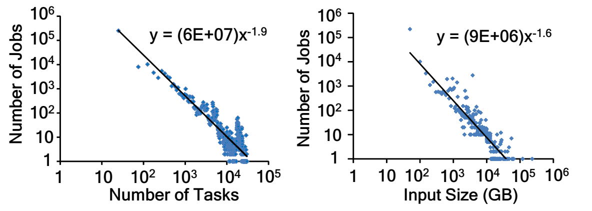
When the input of this job is cached, we use I units of cache. In return, the savings in efficiency is $(1-\mu)I$. The savings is obtained by summing the reduction in completion time across all the tasks in the wave, *i.e.,* $w \cdot (1-\mu)\left(\frac{I}{w}\right)$. Every memory local task contributes to improvement in efficiency. Further, the savings of $(1-\mu)I$ is obtained on every access of the file, thereby making its aggregate value $f(1-\mu)I$ where f is the frequency of access of the file. Hence, the ratio of the benefit to every unit of cache space spent on this job is $f(1-\mu)$, or a function of only the frequency of access of the file. Therefore, *cluster efficiency is best improved by retaining the most frequently accessed files* (LFU-F).

This naturally extends to multi-waved jobs. As jobs in data-intensive clusters typically read entire files, frequency of access of inputs across the different waves of a job is the same. Hence cluster efficiency is best improved by still favoring the frequently accessed files.

To summarize, we have shown that, (*i*) the all-or-nothing property is crucial for improving completion time of jobs as well as efficiency, (*ii*) average completion time is minimized by retaining inputs of jobs with low wave-widths, and (*iii*) cluster efficiency is maximized by retaining the frequently used inputs. We next show some relevant characteristics from production workloads, before moving on to explain the details of PACMan.

3 Workloads in Production Clusters

In this section, we analyze traces from two production clusters, each consisting of thousands of machines – Facebook's Hadoop cluster and Microsoft Bing's Dryad cluster. Together, they account over half a million jobs processing more than 400PB of data. Table 1 lists the relevant details of the traces and the clusters.



(a) Number of tasks (b) Input Size

Figure 6: **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.**

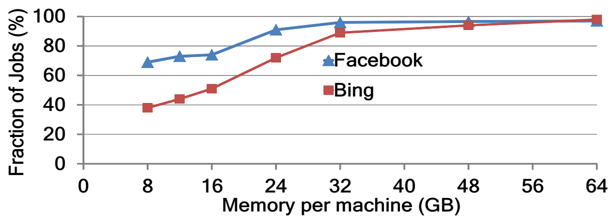


Figure 7: **Fraction of active jobs whose data fits in the aggregate cluster memory, as the memory per machine varies.**

All three clusters co-locate storage and computation. The distributed file systems in these clusters store data in units of *blocks*. Every task of a job operates 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.

Our objective behind analyzing these traces is to highlight characteristics – heavy tail distribution of input sizes of jobs, and correlation between file size and popularity – that are relevant for LIFE and LFU-F.

3.1 Heavy-tailed Input Sizes of Jobs

Datacenter jobs exhibit a heavy-tailed distribution of input sizes. Workloads consist of many small jobs and relatively few large jobs. In fact, 10% of overall data read is accounted by a disproportionate 96% and 90% of the smallest jobs in the Facebook and Bing workloads. As Figure 6 shows, job sizes – input sizes and number of tasks – indeed follow a power-law distribution, as the log-log plot shows a linear relationship.

The skew in job input sizes is so pronounced that a large fraction of active jobs can simultaneously fit their entire data in memory.[3] We perform a simple simulation

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

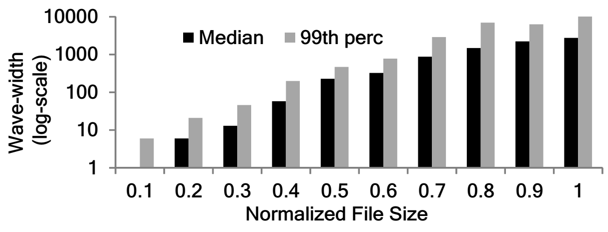


Figure 8: **Wave-width, *i.e.,* number of simultaneous tasks, of jobs as a function of sizes of files accessed. File sizes are normalized to the largest file; the largest file has size 1.**

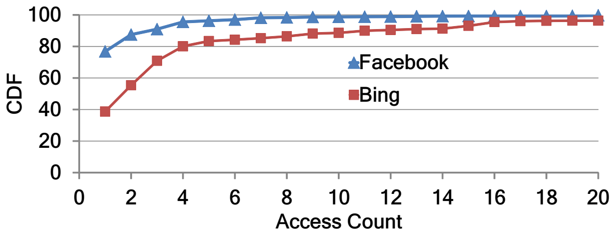


Figure 9: **Skewed popularity of data. CDF of the access counts of the input blocks stored in the cluster.**

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 7 plots the results of the simulation. For the workloads from Facebook and Bing, we see that 96% and 89% of the active jobs respectively can have their data entirely fit in memory, given an allowance of 32GB memory per server for caching. This bodes well for satisfying the all-or-nothing constraint of jobs, crucial for the efficacy of LIFE and LFU-F.

In addition to being easier to fit a small job's input in memory, its wave-width is smaller. In our workloads, wave-widths roughly correlate with the input file size of the job. Figure 8 plots the wave-width of jobs binned by the size of their input files. Small jobs, accessing the smaller files, have lower wave-widths. This is because, typically, small jobs do not have sufficient number of tasks to utilize the slots allocated by the scheduler. This correlation helps to explore an approximation for LIFE to use file sizes instead of estimating wave-widths (§4.2).

3.2 Large Files are Popular

Now, we look at popularity skew in data access patterns. As noted in prior work, the popularity of input data is skewed in data-intensive clusters [15]. A small fraction of the data is highly popular, while the rest is accessed less frequently. Figure 9 shows that the top 12% of popular data is accessed 10× more than the bottom third in the Bing cluster. The Facebook cluster demonstrates a similar skew. The top 5% of the blocks are seven times more popular than the bottom three-quarters.

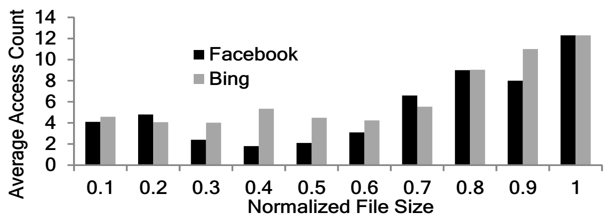


Figure 10: **Access count of files as a function of their sizes, normalized to the largest file; largest file has size 1. Large files, accessed by production jobs, have higher access count.**

Interestingly, large files have higher access counts (see Figure 10). 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. The popularity of large files, whose jobs consume most resources, strengthens the idea from §2.4 that favoring frequently accessed files is best for cluster efficiency.

We also observe repeatability in the data accesses. *Single-accessed files are spread across only* 11% *and* 6% *of jobs* in the Facebook and Bing workloads. Even in these jobs, not all the data they access is singly-accessed. Hence, we have sufficient repeatability to improve job performance by caching their inputs.

4 PACMan: System Design

We first present PACMan's architecture that enables the implementation of the sticky policy, and then discuss the details involved in realizing LIFE and LFU-F.

4.1 Coordination Architecture

PACMan globally coordinates access to its caches. Global coordination ensures that a job's different input blocks, distributed across machines, are viewed in unison to satisfy the all-or-nothing constraint. To that end, the two requirements from PACMan are, (*a*) support queries for the set of machines where a block is cached, and (*b*) mediate cache replacement globally across the machines.

PACMan's architecture consists of a central *coordinator* and a set of *clients* located at the storage nodes of the cluster (see Figure 11). Blocks are added to the PACMan clients. PACMan clients update the coordinator when the state of their cache changes (*i.e.,* when a block is added

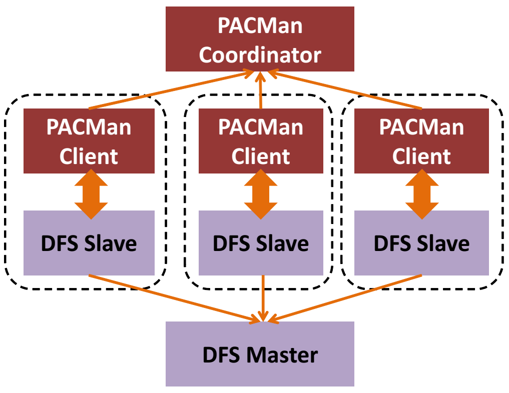


Figure 11: PACMan **architecture. The central *coordinator* manages the distributed *clients*. Thick arrows represent data flow while thin arrows denote meta-data flow.**

or removed). The coordinator uses these updates to maintain a mapping between every cached block and the machines that cache it. As part of the map, it also stores the file that a block belongs to and the wave-width of jobs when accessing that file (§4.2). This global map is leveraged by LIFE and LFU-F in implementing the sticky policy to look for incomplete files. Frameworks work with the coordinator to achieve memory locality for tasks.

The client's main role 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 replicas in cache, which in turn increases the chances of achieving memory locality especially when there are hotspots due to popularity skew [15]. Memory local tasks contact the local PACMan client to check if its input data is present. If not, they fetch it from the distributed file system (DFS). If the task reads data from the DFS, it puts it in cache of the local PACMan client and the client updates the coordinator about the newly cached block. Data flow is designed to be local in PACMan as remote memory access could be constrained by the network.

Fault Tolerance: The coordinator's failure does not hamper the job's execution as data can always be read 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 their cache. The secondary coordinator uses these updates to construct the global cache view. Clients do not update their cache when the coordinator is down.

Scalability: Nothing precludes distributing the central coordinator across different machines to avoid having it be a bottleneck. We have, however, found that the scalability of our system suffices for our workloads (see §5.6).

4.2 Wave-width

Wave-width is important for LIFE as it aims to retain inputs of jobs with lower wave-widths. However, both defining and calculating wave-widths is non-trivial because tasks do not strictly follow wave boundaries. Tasks get scheduled as and when previous tasks finish and slots become available. This makes modeling the tasks that run in a wave complicated. Slots also open up for scheduling when the scheduler allots extra slots to a job during periods of low utilization in the cluster. Therefore, wave-widths are not static during a job's execution. They are decided based on slot availabilities, fairness restrictions and scheduler policies. Unlike MIN, which is concerned only with the *order* in which requests arrive, our setting requires knowing the exact time of the request, which in turn requires estimating the speed-up due to memory locality for each task. All these factors are highly variable and hard to model accurately.

Given such a fluid model, we propose the following approximation. We make periodic measurements of the number of concurrent tasks of a job. When a job completes, we get a set of values, $\langle(w, t(w))\rangle$, such that $0 < t(w) \leq 1$. For every value of wave-width, w, $t(w)$ shows the fraction of time spent by the job with that wave-width. We take measurements every 1s in practice.

Note that while $\sum t(w) = 1$, $\sum w$ is not necessarily equal to the number of tasks in the job. This is because wave boundaries are not strict and tasks overlap between measurements. This led us to drop the idea of using the measurements of $\langle(w, t(w))\rangle$ to divide blocks of a file into different *waves*. Also, such an explicit division requires the scheduler to collectively schedule the tasks operating on a wave. Therefore, despite the potential benefits, to sidestep the above problems, we assign a single value for the wave-width of a file. We define the wave-width of a file as a weighted average across the different observed wave-widths, $\sum w \cdot t(w)$. The wave-width is included in the mapping maintained by the coordinator.

A noteworthy approximation to wave-widths is to simply consider small and large jobs instead, based on their input sizes. As Figure 8 showed, there is a correlation between input sizes of jobs and their wave-widths. Therefore, such an approximation mostly maintains the relative ordering between small and large waves despite approximating them to small and large job input sizes. We evaluate this approximation in §5.3.

4.3 LIFE and LFU-F within PACMan

We now describe how LIFE and LFU-F are implemented inside PACMan's coordinated architecture.

The coordinated infrastructure's global view is fundamental to implementing the sticky policy. Since LIFE

```
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()
    if f == null then                              ▷ Only complete files left
        f = cFiles.getLargestCompleteFile()
    return f.name                                  ▷ File to evict

procedure FILETOEVICT_LFU-F(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.getLeastAccessedIncompleteFile()
    if f == null then                              ▷ Only complete files left
        f = cFiles.getLeastAccessedCompleteFile()
    return f.name                                  ▷ File to evict

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

Pseudocode 1: **Implementation of LIFE and LFU-F – from the perspective of the PACMan coordinator.**

and LFU-F are global cache replacement policies, they are implemented at the coordinator. Pseudocode 1 describes the steps in implementing LIFE and LFU-F. 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 LFU-F).

To make this decision, LIFE first checks whether the client's machine caches the blocks of any incomplete file. If there are many such incomplete files, LIFE picks the one with the largest wave-width 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 one with the largest wave-width. This increases the probability of multiple small files being cached in future.

LFU-F rids the cache of less frequently used files. It assumes that the future accesses of a file is predicted by its current frequency of access. To evict a block, it first checks if there are incomplete files and picks the *least accessed* among them. If there are no incomplete files, it picks the complete file with the smallest access count.

To avoid cache pollution with files that are never evicted, we also implement a window based aging mechanism. Before checking for incomplete and complete files, both LIFE and LFU-F 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 aged and less popular blocks. In practice, we set the window to be large (*e.g.,* hours), and it has had limited impact on most workloads.

PACMan operates in conjunction with the DFS. However, in practice, they are insulated from the job identifiers that access them. Therefore, we approximate the policy of maximizing the number of whole job inputs to maximizing the number of whole *files* that are present in cache, an approximation that works well (§5.3).

Finally, upon caching a block, a client contacts the coordinator by calling ADD(). This allows the coordinator to maintain a global view of the cache, including the access count for every file for implementing LFU-F. 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.

Pluggable policies: PACMan's architecture is agnostic to replacement algorithms. Its global cache view can support any replacement policy that needs coordination.

5 Evaluation

We built PACMan and modified HDFS [2] to leverage PACMan's caching service. The prototype is evaluated on a 100-node cluster on Amazon EC2 [1] using workloads derived from the Facebook and Bing traces (§3). To compare at a larger scale against a wider set of idealized 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.

5.1 Setup

Workload: Our workloads are derived from the Facebook and Bing traces described in §3, representative 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 load proportional to the original clusters.

We meet these goals as follows. We replay jobs with the same inter-arrival times and input files as in the original workload. However, we scale down the file sizes proportionately to reflect the smaller size of our cluster and, consequently, reduced aggregate memory. Thereby, we

Bin	Tasks	% of Jobs		% of Resources	
		Facebook	Bing	Facebook	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.**

ensure that there is sufficient memory for the same fraction of jobs' input as in the original workload. Overall, this helps us mimic the load experienced by the original clusters as well as the access patterns of files. We confirmed by simulation (described shortly) that performance improvements with the scaled down version matched that of the full-sized cluster.

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 is allotted 20GB of cache per machine; we evaluate PACMan's sensitivity to cache space in §5.5.

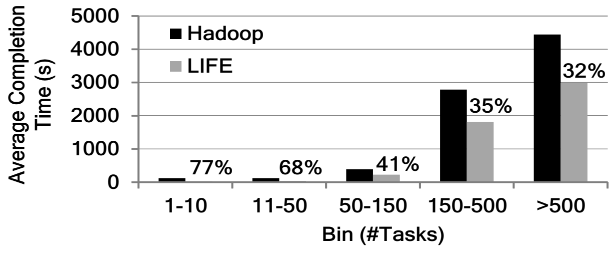
Trace-driven Simulator: We use a trace-driven simulator to evaluate PACMan at larger scales and longer durations. The simulator performed a detailed and faithful replay of the task-level traces of Hadoop jobs from Facebook and Dryad jobs from Bing. It preserved the read/write sizes of tasks, replica 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 cache replacement schemes like MIN.

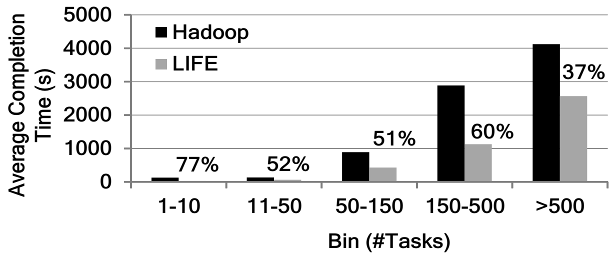
Metrics: We evaluate PACMan on two metrics that it optimizes – average completion time of jobs and efficiency of the cluster. The baseline for our deployment is Hadoop-0.20.2. The trace-driven simulator compared with currently deployed versions of Hadoop and Dryad.

Job Bins: 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 ≤ 10 tasks. The Bing workload, on the other hand, has the corresponding fraction to be smaller but still sizable at 43%. When viewed by the resources consumed, we obtain a different picture. Large jobs (bin-5), that are only 1% and 2% of all jobs, consume a disproportionate 76% and 55% of all resources. The skew between small and large jobs is higher in the Facebook workload than in the Bing workload.

The following is a summary of our results.



(a) Facebook Workload



(b) Bing Workload

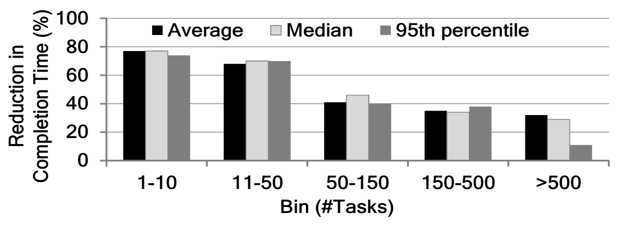
Figure 12: **Average completion times with LIFE, for Facebook and Bing workloads. Relative improvements compared to Hadoop are marked for each bin.**

- Average completion times improve by 53% with LIFE; small jobs improve by 77%. Cluster efficiency improves by 54% with LFU-F (§5.2).
- Without the *sticky* policy of evicting from incomplete files, average completion time is 2× more and cluster efficiency is 1.3× worse (§5.3).
- LIFE and LFU-F are better than MIN in improving job completion time and cluster efficiency, despite a lower cache hit-ratio (§5.4).

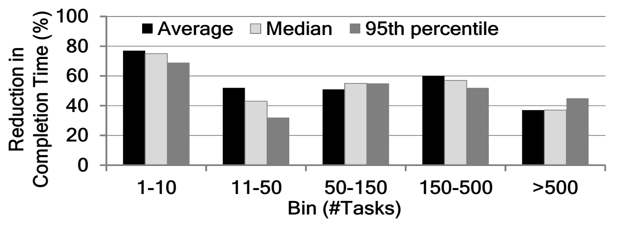
We evaluate our replacement schemes before describing the impact of systemic factors (§5.5 and §5.6).

5.2 PACMan's Improvements

LIFE improves the average completion time of jobs by 53% and 51% in the two workloads. As Figure 12 shows small jobs (bin-1 and bin-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 the sticky policy, 74% of jobs in the Facebook workload and 48% of jobs in the Bing workload meet the all-or-nothing constraint, *i.e.,all* their tasks being memory local. Large jobs benefit too (bin-5) seeing an improvement of 32% and 37% in the two workloads. This highlights LIFE's automatic adaptability. While it favors small jobs, the benefits automatically spill over to large jobs when there is spare cache space.



(a) Facebook Workload



(b) Bing Workload

Figure 13: **Distribution of gains for Facebook and Bing workloads. We present the improvement in average, median and 95th percentile completion times.**

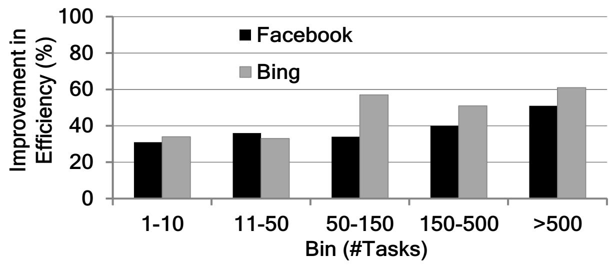


Figure 14: **Improvement in cluster efficiency with LFU-F compared to Hadoop. Large jobs contribute more to improving efficiency due to their higher frequency of access.**

Testbed	Scale	LIFE		LFU-F	
		Facebook	Bing	Facebook	Bing
EC2	100	53%	51%	47%	54%
Simulator	1000′*s**	55%	46%	43%	50%

* *Original cluster size*

Table 3: **Summary of results. We list improvement in completion time with LIFE and cluster efficiency with LFU-F.**

Figure 13 elaborates on the distribution – median and 95th percentile completion times – of LIFE's gains. The encouraging aspect is the tight distribution in bin-1 with LIFE where the median and 95th percentile values differ by at most 6% and 5%, respectively. Interestingly, the Facebook results in bin-2 and the Bing results in bin-3 are spread tighter compared to the other workload.

LFU-F improves cluster efficiency by 47% with the Facebook workload and 54% with the Bing workload. In large clusters, this translates to significant room for executing more computation. Figure 14 shows how this gain in efficiency is derived from different job bins. Large jobs have a higher contribution to improvement in efficiency than the small jobs. This is explained by the observation in §3.2 that large files are more frequently accessed.

An interesting question is the effect LIFE and LFU-F have on the metric that is not their target. With LIFE in deployment, cluster efficiency improves by 41% and 43% in the Facebook and Bing workloads. These are comparable to LFU-F because of the power-law distribution of job sizes. Since small jobs require only a little cache space, even after favoring their inputs, there is space remaining for the large (frequently accessed) files. However, the power-law distribution results in LFU-F's poor performance on average completion time. Average completion time of jobs improves by only 15% and 31% with LFU-F. Favoring the large frequently accessed files leaves insufficient space for small jobs, whose improvement has the highest impact on average completion time.

Overall, we see that memory local tasks run 10.8× faster than those that read data from disk.

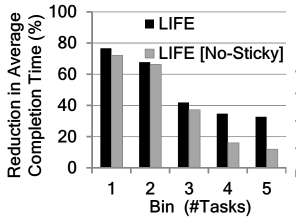
Simulation: We use the trace-driven simulator to assess PACMan's performance on a larger scale of thousands of machines (same size as in the original clusters). The simulator uses 20GB of memory per machine for PACMan's cache. LIFE improves the average completion times of jobs by 58% and 48% for the Facebook and Bing workloads. LFU-F's results too are comparable to our EC2 deployment with cluster efficiency improving by 45% 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 3 shows a comparative summary of the results.

5.3 LIFE and LFU-F

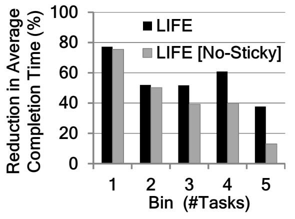
In this section, we study different aspects of LIFE and LFU-F. The aspects under focus are the sticky policy, approximation of wave-widths to file size, and using whole file inputs instead of whole job inputs.

Sticky Policy: An important aspect of LIFE and LFU-F is its sticky policy that prefers to evict blocks from already incomplete files. We test its value with two schemes, LIFE[No-Sticky] and LFU-F[No-Sticky]. LIFE[No-Sticky] and LFU-F[No-Sticky] are simple modifications to LIFE and LFU-F to not factor the incompleteness while evicting. LIFE[No-Sticky] evicts the file with the largest wave-width in the cache, LFU-F[No-Sticky] just evicts blocks from the least frequently accessed file. Ties are broken arbitrarily.

Figure 15 compares LIFE[No-Sticky] with LIFE. Large jobs are hurt most. The performance of LIFE[No-Sticky] is 2× worse than LIFE in bin-4 and 3× worse in bin-5. Interestingly, jobs in bin-1 are less affected. While

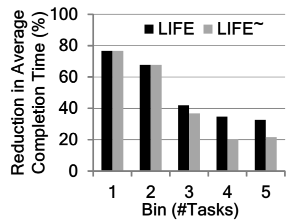


(a) Facebook

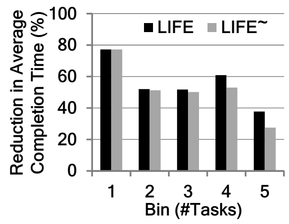


(b) Bing

Figure 15: **Sticky policy. LIFE [No-Sticky] evicts the largest file in cache, and hence is worse off than LIFE.**



(a) LIFE∼ vs. LIFE – Facebook



(b) LIFE∼ vs. LIFE – Bing

Figure 16: **Approximating LIFE to use file sizes instead of wave-widths. Accurately estimating wave-widths proves important for large jobs.**

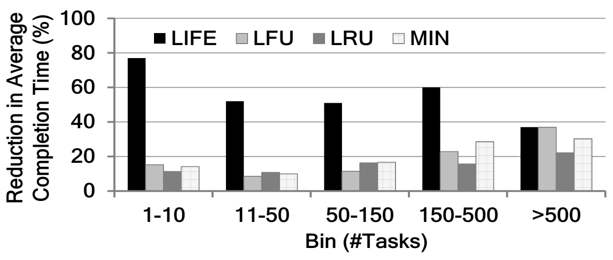
LIFE and LIFE[No-Sticky] differ in the way they evict blocks of the large files, there are enough large files to avoid disturbing the inputs of small jobs.

LFU-F[No-Sticky]'s improvement in efficiency is 32% and 39% for the two workloads, sharply reduced values in contrast to LFU-F's 47% and 54% with the Facebook and Bing workloads. These results strongly underline the value of coordinated replacement in PACMan by looking at global view of the cache.

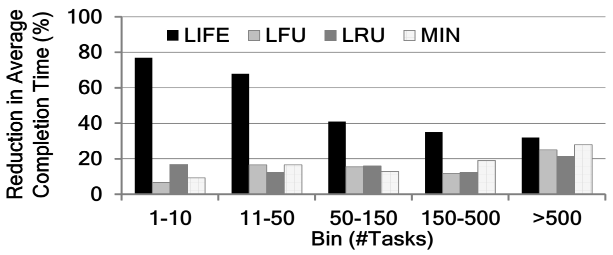
Wave-width vs. File Sizes: As alluded to in §4.2, we explore the benefits of using file sizes as a substitute for wave-width. The intuition is founded on the observation in §3.1 (Figure 8) that wave-widths roughly correlate with file sizes. We call the variant of LIFE that uses file sizes as LIFE∼. The results in Figure 16 shows that while LIFE∼ keeps up with LIFE for small jobs, there is significant difference for the large jobs in bin-4 and bin-5. The detailed calculation of the wave-widths pays off with improvements differing by 1.7× for large jobs.

Whole-jobs: Recall from §2.2 and §4.3 that our desired policy is to retain inputs of as many whole job inputs 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 LFU-F approximate this by retaining as many whole *files* as possible.

Evaluation shows that LIFE is on par with the eviction



(a) Facebook Workload



(b) Bing Workload

Figure 17: **Comparison between LIFE, LFU-F, LFU, LRU and MIN cache replacements.**

policy that retains whole job inputs, for small jobs. This is because small jobs typically tend to operate on single files, therefore the approximation does not introduce errors. For larger jobs, that access multiple files, LIFE takes a 11% hit in performance. The difference due to LFU-F using whole files instead of whole job inputs is just a 2% drop in efficiency. The comparable results make us conclude that the approximation is a reasonable trade-off for the significant implementation ease.

5.4 Traditional Cache Replacement

Our prototype also implements traditional cache replacement techniques like LRU and LFU. Figure 17 and Table 4 compare LIFE's performance. Table 5 contrasts LFU-F's performance. LIFE outperforms both LRU and LFU for small jobs while achieving comparable performance for large jobs. Likewise, LFU-F is better than LRU and LFU in improving cluster efficiency.

Interestingly, LIFE and LFU-F outperform even MIN [11], the optimal replacement algorithm for cache hit-ratio. MIN deletes the block that is to be accessed farthest in the future. As Figure 17 shows, not taking the all-or-nothing constraint of jobs into account hurts MIN, especially with small jobs. LIFE is 7.1× better than MIN in bin-1 and 2.5× better in bin-3 and bin-4. However, MIN's performance for bin-5 is comparable to LIFE. As Table 5 shows, the sticky policy also helps in LFU-F outperforming MIN in improving cluster efficiency.

Overall, this is despite a lower cache hit-ratio. This

Scheme	Facebook		Bing	
	% Job Saving	Hit Ratio (%)	% Job Saving	Hit Ratio (%)
LIFE	53%	43%	51%	39%
MIN	13%	63%	30%	68%
LRU	15%	36%	16%	34%
LFU	10%	47%	21%	48%

Table 4: **Performance of cache replacement schemes in improving average completion times. LIFE beats all its competitors despite a lower hit-ratio.**

Scheme	Facebook		Bing	
	% Cluster Efficiency	Hit Ratio (%)	% Cluster Efficiency	Hit Ratio (%)
LFU-F	47%	58%	54%	62%
MIN	40%	63%	44%	68%
LRU	32%	36%	23%	34%
LFU	41%	47%	46%	48%

Table 5: **Performance of cache replacement schemes in improving cluster efficiency. LFU-F beats all its competitors despite a lower hit-ratio.**

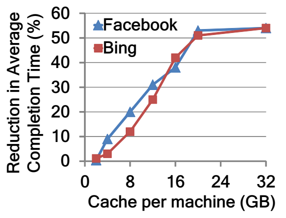
underscores the key principle and differentiator in LIFE and LFU-F – coordinated replacement implementing the sticky policy, as opposed to simply focusing on hit-ratios.

5.5 Cache Size

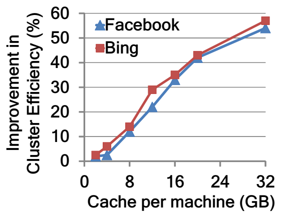
We now evaluate PACMan's sensitivity to available cache size (Figure 18) by varying the budgeted cache space at each PACMan client varies between 2GB and 32GB. The encouraging observation is that both LIFE and LFU-F react gracefully to reduction in cache space. As the cache size reduces from 20GB on to 12GB, the performance of LIFE and LFU-F under both workloads hold to provide appreciable reduction of 35% in completion time and 29% improvement in cluster efficiency, respectively.

For lower cache sizes ($\leq$ 12GB), 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 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 sharper drop for caches sizes $\leq$ 12GB for the Bing workload.

LFU-F reacts more smoothly to decrease in cache space. Unlike job completion time, cluster efficiency improves even with incomplete files; the sticky policy helps improve it. The correlation between the frequency of access and size of files (§3.2), coupled with the fact that the inputs of the large jobs are bigger in the Facebook workload than the Bing workload, leads to LFU-F's performance deteriorating marginally quicker with reducing cache space in the Facebook workload.

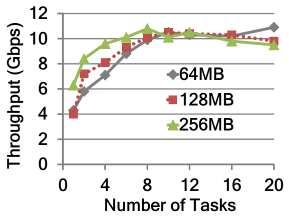


(a) LIFE

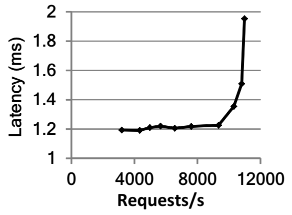


(b) LFU-F

Figure 18: **LIFE's and LFU-F's sensitivity to cache size.**



(a) PACMan Client



(b) PACMan Coordinator

Figure 19: **Scalability. (a) Simultaneous tasks serviced by client, (b) Simultaneous client updates at the coordinator.**

5.6 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. We measure the throughput when tasks communicate with the client and latency when clients deal 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 task reads a block from the client's cache. Figure 19a 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 tasks 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 19b shows, the latency experienced by the requests stays constant at $\sim$1.2ms until 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.

6 Enhancements to PACMan

We now discuss two outstanding issues with PACMan that can help improve its performance.

6.1 Optimal Replacement for Parallel Jobs

An unanswered question is the optimal cache replacement strategy to minimize average completion time of parallel jobs or maximize cluster efficiency. The optimal algorithm picks that block for replacement whose absence hurts the least when the entire trace is replayed. Note the combinatorial explosion as a greedy decision for each replacement will not be optimal. We outline the challenges in formulating such an oracular algorithm.

The completion time of a job is a function of when its tasks get scheduled, which in turn is dependent on the availability of compute slots. An aspect that decides the availability of slots for a job is its fair share. So, when executing tasks of a job finish, slots open up for its unscheduled tasks. Modeling this requires knowing the speed-up due to memory locality but that is non-trivial because it varies across tasks of even the same job. Further, scheduler policies may allow jobs to use some extra slots if available. Hence one has to consider scheduler policies on using extra slots as well as the mechanism to reclaim those extra slots (*e.g.,* killing of running tasks) when new jobs arrive. Precise modeling of the speed-ups of tasks, scheduler policies and job completion times will help formulate the optimal replacement scheme and evaluate room for improvement over LIFE and LFU-F.

6.2 Pre-fetching

A challenge for any cache is data that is accessed only once. While our workloads have only a few jobs that read such singly-accessed blocks, they nonetheless account for over 30% of all tasks. Pre-fetching can help provide memory locality for these tasks.

We consider two types of pre-fetching. First, as soon as a job is submitted, the scheduler knows its input blocks. It can inform the PACMan coordinator which can start pre-fetching parts of the input that is not in cache, especially for the later waves of tasks. This approach is helpful for jobs consisting of multiple waves of execution. This also plays nicely with LIFE's policy of favoring small single-waved jobs. The blocks whose files are absent are likely to be those of large multi-waved jobs. Any absence of their input blocks from the cache can be rectified through pre-fetching. Second, recently created data (*e.g.,* output of jobs or logs imported into the file system) can be pre-fetched into memory as they are likely to be accessed in the near future, for example, when there are a chain of jobs. We believe that an investigation and application of different pre-fetching techniques will further improve cluster efficiency.

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 dealing with parallel jobs that led to a coordinated system that improved job completion time and cluster efficiency, as opposed to hit-ratio.

RAMCloud [18] and prior work on databases such as MMDB [16] 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.

Global memory systems such as the GMS project [5], NOW project [6] and others [14] use the memory of a remote machine instead of spilling to disk. Based on the vast difference between local memory and network throughputs, PACMan's memory caches only serves tasks on the local node. However, nothing in the design precludes adding a global memory view. Crucially, PACMan 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, 12, 23]. Request hit-ratios are best optimized by retaining small files [26], a notion we borrow. We build up on it by addressing the added challenges in data-intensive clusters. Our distributed setting, unlike web caches, necessitate coordinated replacement. Also, we identify benefits for partial cache hits, *e.g.,* large jobs that benefit with partial memory locality. This leads to more careful replacement like 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.

LIFE's policy is analogous to servicing small requests in queuing systems, *e.g.,* web servers [19]. In particular, when the workload is heavy-tailed, giving preference to small requests hardly hurts the big requests.

Distributed filesystems such as Zebra [17] and xFS [8] developed for the Sprite operating system [22] 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 [25] and Spark [21] 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 coordinated caching system for data-intensive parallel jobs. Parallel jobs run multiple tasks simultaneously in a wave, and have the all-or-nothing property, *i.e.,* a job is sped up only when inputs of all such parallel tasks are cached. By globally coordinating access to the distributed caches, PACMan ensures that a job's different tasks, distributed across machines, obtain memory locality. On top of its coordinated infrastructure, PACMan implements two cache replacement policies – LIFE and LFU-F – that are designed to minimize average completion time of jobs and maximize efficiency of the cluster. We have evaluated PACMan using a deployment on EC2 using production workloads from Facebook and Microsoft Bing, along with extensive trace-driven simulations. PACMan reduces job completion times by 53% and 51% (small interactive jobs improve by 77%), and improves efficiency by 47% and 54%, respectively. LIFE and LFU-F outperform traditional replacement schemes, including MIN.

Acknowledgments

We thank Facebook and Microsoft Bing for access to job traces from their production clusters. We are also grateful to our shepherd, Eddie Kohler, and the anonymous reviewers whose inputs greatly improved the paper. For feedback on the draft, we thank Arka Bhattacharya, Mosharaf Chowdhury, Tathagata Das, Aurojit Panda, Ariel Rabkin and David Zats of the AMP Lab at Berkeley, and Sriram Rao of Yahoo! Research.

References

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/instance-types/`.

[2] Hadoop Distributed File System. `http://hadoop.apache.org/hdfs`.

[3] Hadoop Slowstart. `https://issues.apache.org/jira/browse/MAPREDUCE-1184/`.

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

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

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[14] M. J. Franklin, M. J. Carey, and M. Livny. Global Memory Management in Client-Server Database Architectures. In *VLDB*, 1992.

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

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

[17] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *ACM SOSP*, 1993.

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

[19] M. Harchol-Balter M. E. Crovella, R. Frangioso. Connection Scheduling in Web Servers. In *USENIX USITS*, 1999.

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

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

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

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

[24] 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.

[25] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *USENIX OSDI*, 2010.

[26] 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.