Clotho: An Elastic MapReduce Workload/Runtime Co-design

Weiming Shi and Bo Hong
Georgia Institute of Technology
{weimingshi,bohong}@gatech.edu

ABSTRACT
The resource management of a multi-tenant MapReduce cluster can be hard given unpredictable user demands. Conventional resource management scheme would inevitably create a fair amount of spare resource fragments in the system. On the other hand, MapReduce workloads are prone to have a bottleneck stage in the execution pipeline.

To address these two issues under a coherent framework, this paper presents Clotho, a MapReduce workload and runtime co-design that can opportunistically utilize the spare resource fragments in the system to alleviate the bottleneck stage of MapReduce workloads while honoring the SLAs of existing systems. We describe the design and the implementation of Clotho, evaluate it with benchmarks drawn from real MapReduce applications, and demonstrate that it can effectively utilize the spare CPU resource fragments and meanwhile improve the performance of user programs if there is potential to speed up the bottleneck stage of the entire MapReduce execution pipeline.

Categories and Subject Descriptors
H.3.4 [INFORMATION STORAGE AND RETRIEVAL]: Systems and Software—Distributed systems; K.6.4 [Management of Computing and Information Systems]: System Management—Quality assurance

General Terms
Performance, Management

Keywords
MapReduce, Workload/Runtime Co-design, Elastic Resource Management and Performance

1. INTRODUCTION
The need to extract useful information from a gigantic amount of unstructured data has given rise to MapReduce programming model and its data-parallel execution runtime. Hadoop, an open-source MapReduce execution runtime implementation has been widely adopted by companies sharing the same need[1]. Taming the big data monster requires a large-scale computing infrastructure. Efficient resource management for a multi-tenant MapReduce cluster is an imperative but hard problem given unpredictable user demands. If the system is under-provisioned, a fair amount of underutilized system resources would become the resource fragments and lead to a low return on investment (ROI). On the other hand, if the system is over-provisioned, the service level agreements (SLAs) would be prone to defaults. In the face of this dilemma, most resource management schemes in the production have chosen to sacrifice a higher system utilization for a better user experience, which inevitably results in a fair amount of resource fragments in the system.

In this paper, we retrospect over the mindset of the existing resource management schemes of a multi-tenant MapReduce cluster and present our two-cent vision towards a user/system cooperative scheme on addressing this problem. In the context of our research, we refer to the MapReduce user programs or workloads as the users, and refer to the MapReduce execution framework or runtime as the system.

Existing resource management schemes exhibit a bimodal pattern which have basically focused on the two ends of the whole design space spectrum. On the one end is the system-friendly scheme where the resource management decision is solely based on the system capacity and the static estimation of the user demand only available before the execution. Both the simple coarse-grained slot-based scheme of Hadoop (a.k.a MRv1) and the more elaborated fine-grained resource-based scheme of Yarn (a.k.a Hadoop MRv2)[2] are examples of this category. They rely on the user-specified demand for resource management. On the other end is the user-friendly scheme where user programs are treated as black-boxes. Users do not need to specify or update any resource demand during the execution of the user programs. The hard work of getting actual resource demand is left to the execution runtime. The system usually conducts a dynamic calibration during the life cycle of the user programs. Some more elaborated schemes fall into this category [3, 4, 5].

Besides, previous mindset has hinged on the availability of either user demand or system instrumentation and mostly focused on dynamic resource provisioning. It does not consider the resource utilization of the workload as a tangible control knob for the optimization purpose.

We argue that whichever end the resource management scheme stands, it will make it ever harder for either end to achieve a better result because the problem is essentially caused by the lack of communication and cooperation between two parties, i.e.,

1Usually a worst-case estimation
users and system. Efficient resource management of a multi-
tenant cluster should not be a solo of one party but a chorus
of two parties. So for the good of both parties, it is better to
courage joint efforts from both parties instead of leaving the
work to one party alone. A better management decision can be
achieved 1) if the system can dynamically shape the allocated
resource with better clues of the real resource utilization of the
user programs, a.k.a. the Elastic-container approach or 2) if the
users can control the resource utilization of their programs ac-
cording to the allocated resource, a.k.a. the Elastic-workload ap-
proach.

This leads to our proposed co-design scheme which takes
the Elastic-workload approach. We propose a workload/runtime
co-design which installs some lightweight ‘probes’ and ‘knobs’
to Hadoop, an existing MapReduce execution runtime to en-
hance the communication and cooperation between the users
and the system for better resource management purpose. Our
co-design is transparent to the users and honor the resource al-
llocation and job scheduling decisions (and hence the SLAs) of
existing execution runtime system. Besides, it should not de-
grade the performance of running programs. Furthermore, it
should fully explore the data parallelism of the MapReduce work-
loads, reduce the resource fragments in the system and speed
up those MapReduce workloads that can benefit from utilizing
those spare resource fragments.

An analog may help to illustrate our work. If we treat a MapRe-
duce cluster as a big computer and the user-defined Map or
Reduce programs as the instructions supported by this big com-
puter [8], then each MapReduce program can be regarded as a
pipeline of Map and Reduce instructions to be executed, the allo-
cated resource to execute the workload can be regarded as the
distributed execution pipeline of this big computer, and our pro-
posed work can be regarded as a software-hardware co-design
which endeavors to improve both the utilization of the ‘execution
pipelines’ (i.e., the MapReduce cluster) and the performance of
the workload (i.e., the Map/Reduce programs) with the coopera-
tion from both parties.

The primary contributions of this paper are as follows:
• We demonstrate that 1) the resource fragments in the cluster
can be utilized by MapReduce workloads with our proposed
co-design scheme which endeavors to make the resource utiliza-
tion of the MapReduce workloads elastic and that 2) the perfor-
mance improvement can be attained if there is potential to speed up the bottle-neck stage of
the entire execution pipeline by opportunistically utilizing
the resource fragments. Specifically, we have focused on
improving the performance of CPU-intensive map stage
when it becomes the performance bottleneck of the entire
execution pipeline, which is commonly found in MapReduce-
based Monte-Carlo simulations and combinatorial prob-
lem solvers.

• We present Clotho, our preliminary work towards an elas-
tic user/system cooperative resource management scheme.
Clotho features a MapReduce workload/runtime co-design.
It is currently comprised of an elastic map class and a
resource-aware execution runtime. It extends the current
Hadoop runtime with a communication channel between
the user program and the runtime system for resource
management purpose and a light-weight instrumentation
support for detecting map-stage bottleneck in the execu-
tion pipeline of MapReduce programs. It is fully compat-
ible with Hadoop and can dynamically adjust the CPU uti-
ization of user programs to utilize the CPU resource frag-
ments in the system. It can not only improve the resource
utilization of the system but also unleash the performance
potential of the user programs.

The rest of the paper is organized as follows: Section 2 presents
the problem statement and elaborates on the motivation, goals,
challenges and opportunities of our work. Section 3 briefly cov-
ers the related works. Section 4 presents the design and the im-
plementation of Clotho, our proposed workload and runtime co-
design. Section 5 details the experimental evaluation of Clotho.
Finally we conclude the paper and propose our future work in
Section 6.

2. PROBLEM STATEMENT

We observe and argue 1) that a multi-tenant MapReduce cluster
is prone to have a fair amount of resource fragments with the
current resource management schemes; 2) and that MapRe-
duce execution pipeline is prone to have bottleneck stages.

We are seeking for a resource management scheme that uti-
alyze the resource fragments opportunistically in the system to
help alleviate the bottleneck stage of the MapReduce user pro-
grams. Towards that end, we need to enhance the cooperation
between the users and runtime system so that both the sys-
tem utilization and the performance of user programs can be
improved.

2.1 Arguments on Motivation

Firstly, why current system-friendly or user-friendly scheme
has failed to reduce the resource fragments in the system? For
the system-friendly scheme, if user demands are overestimated
and fulfilled by the system, then the resource fragments can
hardly be eliminated. For the user-friendly scheme, if some
large resource fragments are detected as is in the coarse-grain
slot-based scheme, the execution runtime can overprovision as
if there are more slots available to the users. However, with
the emergence of fine-grained resource allocation scheme, the
large headroom left for this coarse-grained dynamic overpro-
visioning to work is diminishing. Besides, there is no guarantee
that this approach can effectively utilize the spare resource on
the right dimension. For instance, when the CPUs are idle but
the I/O is stressed, simply oversubscribing more I/O-intensive
works will not utilize the spare CPU resource but may even de-
grade the performance of existing running programs. By large,
there will inevitably be some unfavorable resource fragments in
the system as long as the real dynamic resource utilization of the
workload is lower than its static worst-case estimation. Besides,
when the system load is light, there will be some idle resource
in the system. We would better find a good use of them too
because they also incur the operational cost.

Secondly, why there will be bottleneck stages in the MapRe-
duce programs? A MapReduce program consists of multiple
consecutive stages and its resource utilization and performance
is not merely determined by only one stage but all the stages in-
volved. Ideally, if every single stage of the MapReduce execution
pipeline is very ‘balanced’, say, in terms of its maximum achiev-
able throughput, then the only way to improve performance fur-
ther is to optimize every stage of the whole pipeline. However,
this perfectly ‘balanced’ scenario hardly happens in the reality.
Because the runtime resource utilization is hard to know before-
hand without profiling and the resource utilization of the system
is also hard to predict given the uncertainty of system load. If a

\[\text{Here ‘large’ means that the spare resource rate is large and the}
\text{idle time window is wide}\]
disproportional amount of work is assigned with a incommensurate amount of allocated resources in each stage of the MapReduce pipeline. Chances are one of the stages may become the bottleneck of the whole execution pipeline as it could not sustain a higher throughput that can be achieved by the other stages. For example, assigning too much CPU-intensive work to one upstream stage may starve its immediate downstream stage as there would not be enough output data generated to serve as the input data for the downstream stage. Failing to balance the maximum achievable throughput of all the stages of the pipeline can cause the entire workflow to block and prolong the execution of the entire workflow, which severely undermines the performance improvement achieved by other stages of the pipeline and degrade the utilization of the underlying computing infrastructure.

Without the cooperation from both users and system, the bottleneck of MapReduce programs can hardly be detected and resource fragments of the system can hardly be utilized to improve the performance of running MapReduce programs. Hence we propose our user/system cooperative resource management scheme to help to address the issues of both MapReduce programs and the MapReduce system under a coherent framework.

2.2 Goals

Existing native operating system or virtualization hypervisor has already had some effective mechanisms for clipping the resource utilization of the user programs or virtual machines. However, in the face of resource fragments, existing user programs and system lacks a light-weight mechanism to utilize it for the benefits of both parties. Clipping the resource utilization is an effective way to enforce performance isolation but it limits the potential for optimization. Being able to utilize spare resource fragments opportunistically is an important but unfortunately missing functionality for the system to achieve a better utilization and for the user to have better performance.

We try to bridge this gap with our Elastic-workload approach by enabling the workload to trade off between its resource utilization and performance based on the allocated resources. Our goal is not only to simply clip but also to expand the resource utilization of the MapReduce workload to leverage the otherwise spare resource fragments in the system if the expansion is possible and favorable to speed up the execution of user programs.

2.3 Challenges and Opportunities

Our co-design approach needs the controllability of the resource utilization for the MapReduce workloads, which is unfortunately missing in the current user programs and runtime system. In the current Hadoop runtime, each stage of the MapReduce execution pipeline has to be served by a fixed number of worker threads. Besides, the fixed number of worker threads needs to be predetermined without any awareness of the online information such as the resource utilization of the user programs and the load of the system. This not only limits the maximum resource utilization of the user programs but also restrains the user programs from exploring the spare resource in the system. For instance, the mapper is not able to adapt its resource utilization to the load of the system by adding more worker threads to leverage underutilized resources or retiring some worker threads to reclaim the resource for other users when the system load becomes high.

---

3. RELATED WORK

Conventional resource allocation scheme of Hadoop (MRv1) is conducted based on slots. A slot presents a certain amount of CPU cores and memory and serves as the unit of resource allocation. A better resource utilization can be achieved if the system distinguishes the workloads and conducts the dynamic admission control based on the actual [3, 4, 5] or dominant resource demand [7] of the workload and the load of the system.

Yarn [2] improves over Hadoop by conducting the resource allocation based on the user-specified (but not the actual) CPU and memory demand. However, it can not dynamically change the amount of resource just like Hadoop. Yarn can also leverage the cgroup of recent Linux kernel to enforce performance isolation among different workloads like Mesos [8].

Our work uses the same mechanism as Elastic Phoenix [9] to dynamically allocate more CPU resource to the workloads. However their work is based on a shared-memory MapReduce implementation and focuses more on the malleable API design. It does not targets for solving the resource fragment and the bottleneck problem as we do.

4. CLOTHO

We find that there is a proposal to take the Elastic-container approach[10]. To the best of our knowledge, we are the first to take the Elastic-workload approach to advocate a user/system cooperative resource management scheme for the MapReduce applications.

4.1 Merits of Clotho’s Design

We chose the Elastic-workload approach as opposed to Elastic-container approach as we believe it has the following merits: 1) It can simplify the resource allocation and reduce the instrumentation overhead as is found in the Elastic-container approach. Because if the allocated resource based on users’ worst-case estimation can be fully utilized, then there will be almost no headroom left for the system to overprovision and hence no deliberate need to monitor the resource utilization of user programs all the time. 2) It is much flexible if we can make existing user programs to utilize the resource fragments that are hard to eliminate entirely by the system; 3) It can potentially speed up the execution of user programs if opportunistically utilizing spare resource fragments helps; 4) It can avoid the extra scheduling overhead introduced by the dynamic overprovisioning and the lack of matching workload to fit the resource fragments; 5) It is orthogonal to the efforts on improving the resource management workflow, the fairness guarantee and the performance isolation.

---

3The default is one single thread; the map stage has Multi-threadedMapper as Hadoop’s multi-threaded support

4Resource monitoring of the entire system is still needed
4.2 User Input Requirement

For Clotho to work, we need the efforts from the user side to make sure that the input data of the map stage is splittable. For input-splittable MapReduce workloads, no change to the input data needs to be done as long as there is enough amount of induced work for each mapper. For some input-unsplittable MapReduce workloads, it is still possible to fulfill the splittable input requirement with the domain knowledge of users. Users need to transform the original single input record into a series of new input records. This essentially requires that the new input records can equivalently induce a same amount of work as the original single record. For instance, \( X \) runs of Monte Carlo simulation is equivalent to two consecutive \( X/2 \) runs of simulation with users’ domain knowledge. So the origin single-record input “\( X \)” can be transformed to be two-record input “\( X/2 \) \( \cup X/2 \)”.

4.3 Overview of the Design of Clotho

Clotho builds upon the Hadoop (stable version 1.1.2) and inherits its master-slave architecture. There is also a plan to integrate our current work to Yarn later. As a part of Clotho, we introduce a new Mapper Class, i.e. ElasticMapper (see § 4.4) to the family of stock Mapper classes. ElasticMapper is designed to control the CPU resource utilization of the map stage. We augment the Child class which executes the actual map/reduce task with a local resource agent (LRA) and the TaskTracker class with a local resource manager i.e. LRM (see § 4.5). LRA and LRM follows the same master-slave architecture of Child and TaskTracker of Hadoop. They establish a communication channel between the user programs and runtime system for resource management purpose.

LRM leverages the performance counters of Hadoop runtime to monitor and collect the latest resource utilization information. LRA heartbeats with LRM to share the information of the user programs with LRM proactively, which is previously missing in current Hadoop runtime. LRA also conducts the bottleneck detection of the user programs with a lightweight instrumentation of key methods of ElasticMapper. With the user program information reported by all the LRAs, LRM coordinates with LRA to trigger the ElasticMapper to spawn/retire worker threads to expand/shrink its CPU resource utilization. ElasticMapper conducts the user program instrumentation and the actual CPU resource expansion and shrinkage. ElasticMapper and LRA follow an observer design pattern with ElasticMapper registers to the LRA for action notification.

4.4 Clotho’s ElasticMapper

Clotho’s ElasticMapper is designed to make the resource utilization of map stage elastic. It features a dynamic-sized thread pool and the light-weight instrumentation support.

The dynamic-sized thread pool can change the number of worker threads dynamically based on the resource utilization of the user programs and the load of the system. The core size of thread pool is exported as a configurable property to the users and can be expanded or shrinked dynamically based on the resource management need.

The instrumentation support leverages our injected ‘probes’ to the methods of ElasticMapper. We use these ‘probes’ to detect whether the user-defined function (map stage) has any bottleneck that can be alleviated by utilizing available spare CPU resource fragments. The map-stage execution flow starts with RecordReader reading in a new record (a key-value pair) from its input split (either on the local or the remote node), continues with Mapper applying the user-defined map function to input record to generate a series of intermediate key-value pairs, and alternates with RecordWriter writing the output key-value pairs to the local file system. This continues until all the key-value pairs in the input split are drained. We instrument 1) the NextKeyValue method of RecordReader, 2) map method of Mapper and 3) write method of Context with three new performance counters to collect the aggregated time spent on these function calls. To be fully compatible with stock Hadoop runtime, we did not change its map-stage code structure where Context.write is embedded inside the Mapper.map function. Some simple math derivations are conducted to get the desired timing information (see § 4.6). We keep the stock RecordReader. NextKeyValue, Mapper.map, Context.write function as one set of functors and our instrumented counterparts as another set of functors. We start with the instrumented functor and switch to the original functor to reduce the runtime overhead of the user programs once the desired timing information is collected. Because we expect to be able to have a good estimation of the proportion of the time spent on each substage after several iterations of map stage, we do not need to instrument the map stage throughout its entire life cycle to detect whether there is a bottleneck in the map stage.

4.5 The Local Resource Manager and Agent

LRA and LRM form a poll-based communication pattern. LRA invokes a RPC call to update LRM with the information of running map tasks using the instrumentation mechanism of ElasticMapper. LRM leverages Hadoop performance counters to collect and keep the state of the CPU utilization of the local host while LRA serves as the agent of LRM to collect the dynamic runtime information for each running child JVM (e.g. the map task). LRA also conducts a cost/benefit analysis to determine whether the mapper execution substage would become the bottleneck of map-stage before reporting the result to the LRM. When LRA is informed by LRM that there is spare CPU resource on the local host and its cost/benefit analysis favors the expansion, it will notify the child JVM to spawn a new worker thread to take advantage of the spare CPU resource opportunistically to execute the same user-defined map function on an unprocessed input record. Likewise, when LRA is informed that the load is rising and previously spare CPU resource needs to be reclaimed for other user programs, it will notify the child JVM to shrink by setting a flag in RecordReader so that the next time ElasticMapper calls NextKeyValue to request a new input record, it will return false and the worker threads can exit gracefully. The number of working threads is kept in an atomic integer. Once enough threads have exited since the last time a flag to shrink is set, the flag will be unset and the remaining worker threads can continue the execution throughout their life cycle.

4.6 Cost/Benefit Analysis

The cost/benefit analysis is conducted by LRA based on the collected aggregated time spent on each substage. For each iteration of map-stage execution, the record reading and writing part incurs both I/O and CPU resource \(^5\) while the mapper execution is mostly CPU-intensive.

Denote the aggregated time spent on record reading, mapper execution and record writing as \( M_r \), \( M_c \) and \( M_w \) respectively. We define the map-stage to be CPU-expandable (resp. I/O-expandable) if

\[
M_c \gg M_r + M_w \quad (\text{resp. } M_c \ll M_r + M_w),
\]

\(^5\)CPU-intensive serialization/deserialization and I/O operations
i.e., the time spent on CPU-intensive part is much longer than (resp. negligible comparing with) the time spent on I/O-intensive part. Many MapReduce applications exhibit the characteristics of having a CPU-intensive map stage, such as those involving Monte-Carlo simulation and combinatorial optimization where the CPU-intensive map stage is usually the bottleneck of the entire workflow. Examples of I/O-expandable applications include many applications that need to conduct a distributed sorting. There have been some clues to leverage the compression and the fast interconnect techniques to improve the performance of the distributed sorting. We would leave investigating and incorporating I/O-expandable applications into our framework as the future work.

As our ElasticMapper will eventually generate the same intermediate data as the stock Mapper, the downstream stages in the execution pipeline should hardly be affected. Therefore, our cost/benefit analysis is based on the comparison between the original execution time of the map-stage and its predicted counterpart. Assume the user program uses single thread worker to fulfill the work of the map stage which is a common case for most user programs. The original total completion time \( T_o \) is

\[
T_o = M_r + M_c + M_w
\]

The predicted total completion time after expanding the CPU resource utilization is

\[
T_p = \alpha M_r + 1/p \cdot M_c + \alpha M_w + O_p
\]

Here \( p \) is the number of worker threads ElasticMapper spawned, \( O_p \) is the synchronization overhead with \( p \) worker threads, \( \alpha \) is the ratio of the amount of transformed input data to the amount of the original data. \( \alpha = 1 \) if the input data is splittable as no transform is needed and hence no extra amount of data needs to be read. \( \alpha = p \) when the input data would be bloated by at least a factor of \( p \) after transformation so that every extra worker thread has work to do. \( 1/p \) depicts our intention to reduce the execution time \( M_r \) by a factor of \( 1/p \).

Hence we have

\[
T_p - T_o = (\alpha - 1)(M_r + M_c) + O_p - (1 - 1/p)M_c
\]

If \( M_c > \frac{\alpha - 1}{\alpha - \frac{1}{p}}(M_r + M_c) + \frac{1}{p} O_p \), then \( T_p < T_o \), i.e. the resource expansion can help to speed up the map-stage execution. In the reality, the synchronization overhead of \( p \) worker threads is the same order of \( M_r + M_c + M_w \) i.e., \( O_p = \Theta(M_r + M_c + M_w) \). It is easy to check that the CPU-expandable applications which has \( M_c > M_r + M_w \) will favor the CPU resource expansion if spare resource fragments are available.

5. EVALUATION

5.1 Benchmarks and Experimental System

We conduct the experiments on a four-node Cluster. All these four nodes are configure to run the DataNode, TaskTracker, LRM. One of them is also configured to run the NameNode and JobTracker. Each node hosts a 12-core CPU, 48G memory, a 500GB SATA disk (3Gb/s 7200 RPM). We set up a maximum 16 map slots and 8 reduce slots for our experiment system and leave out some spare resource holes deliberately to mimic the possible resource fragments or light load scenario in the real production environment. We verify the correctness of cost/benefit analysis of Clotho with our synthesized benchmark based on WordCount. We evaluate the improvement of the system utilization and speedup of user programs with real MapReduce applications shipped with stock Hadoop and present a representative result from running a modified Distributed-Pentomino. Distributed-Pentomino conducts a CPU-intensive combinatorial search. Our modified DistributedPentomino application need the efforts from user to transform the original unsplittable input record to a series of equivalent input records as we mentioned in Section 4.2 so that the worker threads spawned by ElasticMapper can have enough work to do and hence help to reduce the execution time of the whole application.

5.2 Correctness of Cost/Benefit Analysis

To verify the correctness of our Cost/Benefit Analysis, we create a synthesized benchmark based on Hadoop WordCount example. We pad a varied number of CPU-intensive iterations into the map function of WordCount to tune the level of CPU-intensity of the workload. We run the experiments with the stock WordCount (treated as our baseline) and our ElasticMapper-enabled WordCount. Figure 1a shows the execution time distribution of the map stage. It consolidates four groups of results. Each group \( M(y) \) has three bars \( B(1,y), E(1,y), E(2,y) \) respectively. Here \( B(1,y) \) represents the stock single-threaded WordCount with \( y \) iterations of CPU-intensive padding, \( E(x,y) \) represents the ElasticMapper-enabled WordCount with \( x \) worker threads and \( y \) iterations of CPU-intensive padding.

As we can find out, the majority of the map-stage execution time is spent on record reading and writing for stock WordCount where our ElasticMapper cannot help to reduce the execution time of the whole map stage. However, with the increasing number of CPU-intensive iterations padded to the map function, the portion of the time spent on mapper execution grows and our ElasticMapper starts to help reducing the execution time of the whole map stage.

The per-group speedup improvement shown in Figure 1b. Here \( M(y) \) represents the group with \( y \) iterations of CPU-intensive padding and Elastic\((x)\) represents the ElasticMapper-enabled WordCount with \( x \) worker threads. As we can see, initially when the workload is not much CPU-intensive \((y \text{ is small})\), the performance of ElasticMapper-enabled WordCount is worse than its stock counterpart. This is caused by our instrumentation overhead which we endeavor to reduce further in our future work.
ElasticMapper better performance by reducing the map-slot footprint (or the 100% when the number of spawned worker threads increases observe that the CPU utilization increases from 50% to almost 5.3 Improvement on System Utilization and iterations of CPU-intensive padding is executed by two worker threads.

5.3 Improvement on System Utilization and User-program Speedup

We use Ganglia[11] to monitor the utilization of our cluster and observe that the CPU utilization increases from 50% to almost 100% when the number of spawned worker threads increases from 1 to 4 (see Figure 2a).

Besides, we also find that we can actually achieve an even better performance by reducing the map-slot footprint (or the demand) of user programs. With ElasticMapper, user programs can be written to consolidate the work previously done by multiple map tasks into a single map task and utilize the spare CPU fragments in the system opportunistically. When the system load is low, ElasticMapper can not only use up to almost the same amount of CPU resource as is consumed by multiple concurrent single-threaded map tasks but also reduce the execution delay caused by the scheduling overhead of the execution runtime. In Figure 2b, B(2001) is our baseline with 2001 single-threaded map tasks and E(x, y) is ElasticMapper-enabled with x map tasks and y worker threads per task. As we can figure out that E(501, 4) has the smallest footprint of map slots but it has the best performance. This is because that E(501, 4) has essentially the same amount of worker threads (i.e. 2000 worker threads) as our baseline B(2001) and hence consume the same amount of CPU resource as is consumed by the baseline. Besides, it has less scheduling overhead compared with the baseline. E(501, 4) can even outperform E(2001, 4) although E(2001, 4) has the maximum number of worker threads. This is because it is less likely for ElasticMapper to find any spare CPU resource for use when the system load is high. However, because of its small map-slot footprint, it still incurs less scheduling overhead compared with E(2001, 4).

6. CONCLUSIONS AND FUTURE WORKS

We present Clotho, our preliminary work towards an elastic user/system cooperative resource management framework which targets for reducing the resource fragments in the MapReduce system and meanwhile alleviating the bottleneck in the MapReduce execution pipeline by utilizing these fragments opportunistically. Our Elastic-workload approach has the implications for both the users and the system. For the users, if they want to be able to leverage the spare resources in the system opportunistically, it is better to turn their MapReduce programs to be elastic as is advocated by ElasticMapper. For the system, elastic workload provides the system with the ability to trade off between the system utilization and the user experience.

There are several directions we can extend our works: 1) mitigate the stragglers under our framework; 2) add memory and I/O as the first-class resources into our framework; 3) extend our framework to be a proactive scheme with resource utilization prediction.

7. REFERENCES