REEF: Retainable Evaluator Execution Framework

Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Josh Rosen, Russell Sears, Markus Weimer

{bchun,tcondie,ccurino,cdoug,sergiym,shravan,raghu,sriramra,sears,mweimer} @microsoft.com

bdmyers@cs.washington.edu joshrosen@cs.berkeley.edu

Microsoft Cloud Information Services Laboratory

ABSTRACT

In this demo proposal, we describe REEF, a framework that makes it easy to implement scalable, fault-tolerant runtime environments for a range of computational models. We will demonstrate diverse workloads, including extract-transform-load MapReduce jobs, iterative machine learning algorithms, and ad-hoc declarative query processing. At its core, REEF builds atop YARN (Apache Hadoop 2's resource manager) to provide *retainable* hardware resources with lifetimes that are decoupled from those of computational tasks. This allows us to build persistent (cross-job) caches and cluster-wide services, but, more importantly, supports high-performance iterative graph processing and machine learning algorithms.

Unlike existing systems, REEF aims for *composability* of jobs across computational models, providing significant performance and usability gains, even with legacy code. REEF includes a library of interoperable data management primitives optimized for *communication and data movement* (which are distinct from storage locality). The library also allows REEF applications to access external services, such as user-facing relational databases.

We were careful to decouple lower levels of REEF from the data models and semantics of systems built atop it. The result was two new standalone systems: *Tang,* a configuration manager and dependency injector, and *Wake*, a state-of-the-art event-driven programming and data movement framework. Both are language independent, allowing REEF to bridge the JVM and .NET.

1. INTRODUCTION

As Hadoop has matured, the range of computational primitives expected by its users has broadened. Numerous performance and workload studies have shown that Hadoop MapReduce is a poor fit for iterative computations, such as machine learning and graph processing, and also for the extremely small, ad hoc queries that compose the vast majority of jobs (though *not* the majority of computation) on production clusters.

Hadoop 2 addresses this problem by factoring MapReduce into two components: a MapReduce *application master* that schedules computations for a single job at a time, and *YARN*, a cluster resource manager that coordinates between multiple jobs and tenants. Although YARN allows a wide range of computational frameworks to coexist in one cluster, many challenges remain. REEF builds on YARN to provide features that are crucial to current workloads:

Retainability of hardware resources across tasks and jobs. This allows iterative and workflow computations to perform well.

Composability of operators written for multiple computational frameworks and storage backends. This is crucial for usability, interoperability and performance.

Cost modeling for data movement and single machine parallelism. Hardware and software trends favor these approaches over Hadoop-style storage locality and coarse-grain parallelism.

Fault handling including checkpointing of task state, and deterministic task invocations (which also aids in debugging).

Elasticity REEF's checkpointing and preemption mechanisms allow jobs to adapt as resource allocations change.

Above all else, we seek to improve the ***usability*** of scalable data processing. These unfortunate rules of thumb characterize the current state of the art:

1. Naïve jobs are orders of magnitude slower than tuned ones, and performance issues are rarely application-specific.
2. Applications manually partition jobs across computational runtimes and explicitly convert between wire-level formats.
3. As cluster software is upgraded, "bit-rot" changes job semantics, making it difficult to reproduce old results or perform post mortem analysis of compromised data sets.

In turn, most ***developer time*** (often the key bottleneck) is spent on issues that should be handled by the underlying data management infrastructure. This was our prime target as we designed REEF.

2. RELATED WORK

In this section, we survey specialized computational frameworks with a focus on low-level issues that prevent Hadoop from efficiently supporting such systems. Next, we explain the relationship between cluster resource managers and REEF.

Hadoop MapReduce is hardcoded to use coarse-grained, stateless tasks to run MapReduce jobs, which it implements as a scan, a shuffle (sort) phase, followed by scans over groups of (sorted) data. It achieves fault tolerance by persisting the output of each task to disk. This approach has a number of performance problems. Complex computations are run by chaining MapReduce jobs together into workflows. Each step of the workflow persists its output to disk, incurring disk and communication overheads, as well as the computational overhead of serializing and parsing the

Proceedings of the VLDB Endowment, Vol. 6, No. 12

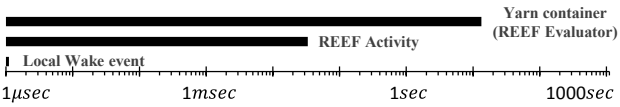

Figure : Task spawn latencies in REEF

intermediate data. These costs often dominate job runtimes. PACMan [1] improves matters by scheduling tasks to make better use of the OS file cache, although it still writes intermediate data to disk. Assuming degraded fault tolerance and scalability is acceptable, the best solution is to entirely avoid serialization and parsing overheads by keeping the parsed data in RAM. A wide range of systems (including some listed below) have used similar tricks to improve performance.

Ultimately, application developers have moved to higher-level programming abstractions (such as Hive and Pig's SQL-like query languages), which currently run as job workflows that implement unintended operations (such as join) atop MapReduce and demand richer storage primitives (indexes) than HDFS provides. Although such jobs run atop Hadoop MapReduce, they do so with significant performance penalties. In response, workflow-friendly Hadoop replacements, such as PACT [2], Dryad [3], and Hyracks [4] have emerged, with support for additional operators, arbitrary computational DAGs and indexing. Although these systems are promising, like Hadoop, they couple resource management and computational models, locking out many new (and existing) workloads.

Mahout is an extreme example of systems that suffer from unnecessarily reparsing input data. It is a machine learning library built atop well-behaved MapReduce jobs. As a consequence it repeatedly reads the same data, and performance suffers accordingly [5]. Systems such as Giraph (for graph processing), and VW (machine learning) move beyond workflows to iterative computations. This highlights another performance problem: Hadoop's coarse task granularity. The cost of invoking a Map or Reduce task in Hadoop is surprisingly high; best practices dictate that each task should run for at least 10's of seconds to a minute. Giraph and VW address this problem by requesting a bunch of Map tasks and then refusing to release them until the job is complete. This causes performance and reliability problems for their jobs and for the cluster as a whole.

In contrast to Hadoop, sub-millisecond tasks are common in traditional task-based environments [6]. Even in such environments, coarse tasks are a common bottleneck that prevent further linear scaling. Clearly, graph-processing and machine-learning systems would benefit from direct support for fine-grained tasks and low-latency communication.

Cluster managers like YARN, Mesos [7], Condor [8] and others[1] provide raw cluster resources to higher-level applications. They do so in a coarse-grained fashion: as in Hadoop, allocation requests should involve at least 10-60 seconds of machine time (Figure 1).

REEF runs atop YARN, though it could be retargeted to most of the other cluster managers. Like the others, YARN introduces a two-level scheduler that decouples resource and job management. This allows MapReduce, workflows and iterative computations to coexist and run efficiently. In addition to providing access to computational resources, YARN exposes storage locality information from the underlying store (currently just HDFS), allowing computational frameworks to reason about cluster and storage topologies as they see fit.

[1] Google Borg and Microsoft Windows Fabric are internal versions of the same concept that do not have citable references.

3. KEY ABSTRACTIONS

REEF is structured around the following key abstractions:

Job Driver: The user-supplied control logic. There is exactly one Driver for each Job. The duration and characteristics of the Job are determined by this module.

Activity: User-supplied logic that performs the data processing. Activities are a generalization of Hadoop's Map and Reduce Tasks. REEF encourages a strict separation between the control plane and the data plane of the computation.

Evaluator: The runtime for Activities. There is a 1:1 mapping between Evaluators and YARN Containers. Evaluators run one activity at a time, but a single evaluator may run many activities throughout its lifetime. This enables sharing among Activities (e.g., caching) and reduces scheduling costs.

Services: Objects and daemon threads that are retained across Activities that run within an Evaluator. Examples include caches of parsed data, intermediate state and network connection pools.

The Driver orchestrates communication and computation, resource management, and fault handling. REEF provides libraries and services that make it easier to implement new Drivers.

4. ACTIVITY LIFECYCLES

Above, we said that Activities are a generalization of Hadoop Tasks. Activities contain arbitrary application code, and are able to behave in arbitrary ways. Here, we describe two variants. The first mimics Hadoop, allowing REEF to run legacy jobs (and, with the help of Services, to improve their performance). The second provides coarse-grained application tasks that may be checkpointed and migrated between Evaluators. This enables fault tolerance and dynamic load balancing.

The most natural use of YARN (and therefore REEF Evaluators) is to request a new container for each task, and to return the container to YARN when the task completes. This provides YARN with the greatest possible flexibility when it comes to sharing cluster resources across jobs and tenants, and mirrors the approach used in earlier versions of Hadoop, making this a reasonable approach for existing jobs. However, container allocations are expensive: they consume scheduler resources, and incur high startup costs.

REEF Evaluators provide two alternatives. The first leverages Evaluators' ability to run multiple Activities, one after the other. By using Services to cache Activity state and outputs, Evaluators can significantly reduce the cost of running chains of Activities.

However, coarse-grained tasks have a number of fundamental drawbacks. Scheduling decisions are complicated by long-running tasks with unpredictable runtimes and job parallelism is limited by the overhead of spawning additional tasks. Furthermore, many iterative and graph processing computations process each message by spawning a new task; with heavyweight tasks, this leads to unacceptable overheads; Evaluator reuse improves the situation considerably by reducing task spawn times (Figure 1).

Alternatively, certain classes of computation (such as simulations) consist of long running tasks that cannot be re-run from scratch; REEF's checkpointing service makes it easy to serialize Activity state to disk and to restart the Activity on a different machine. This allows applications to implement fault tolerance by periodically checkpointing state, and facilitates elastic load balancing and scheduling (idle, or shrinking tasks can be paused or consolidated onto fewer machines) [9]. Of course, in addition to serializing state to disk, checkpointing requires support from various REEF components, including networking and storage.

5. DATA MANAGEMENT SERVICES

REEF's data management service is a set of libraries that provides mechanisms for networking, storage, checkpointing, and other data processing infrastructure. It provides a naming abstraction called an *Identifier* that is the basis for many of these mechanisms. REEF Identifiers are glorified URIs that are associated with resource types, such as files, network addresses and REEF Activities. This level of indirection allows Activity instances to be checkpointed in one Evaluator and restored in another; REEF's network management layer transparently routes the requests as appropriate.

Identifiers also help provide storage independence. REEF's storage manager provides access to generic file and index objects and Identifiers encode both the name of the object and the underlying storage implementation, allowing applications to treat multiple storage systems interchangeably.

These facilities are accessed through two APIs: one in the Driver and one in the Activity. The Driver API requests and configures Services when Evaluators are created, and the Activity API provides access to the actual functionality. This separation makes it easy to add new data management services to REEF.

In addition to communication and storage, the REEF data management APIs include computational primitives. One that bears mentioning is the *combine* primitive, which groups streams of (key, value) pairs by key, invokes a combiner callback on pairs within the same group, and periodically ships combined values to their destination. This is the basis of Hadoop MapReduce's combiner optimization, as well as aggregation trees and many machine learning training algorithms. Various implementations make sense in different scenarios, including in-memory, on-disk and distributed approaches. Providing computational primitives as part of REEF allows us to automatically leverage any advanced storage and networking features of the underlying cluster. We are particularly interested in using Sailfish-style I-Files [10] to build in-network sorting, grouping and aggregation primitives.

6. WAKE: DATA AND CONTROL PLANES

Two challenges immediately arise during the implementation of new computational frameworks. Scalable Driver implementations must coordinate between large numbers of potentially faulty nodes, and handle network timeouts, unexpected messages and so on. Furthermore, to obtain good performance, Drivers must make heavy use of asynchronous operations. We have found that events are a natural fit for such environments.

Similarly, when implementing high-performance dataflow and task-based computations, one wants control over (and visibility into) the flow of work between threads and concurrency between tasks of the same type. With current multi-core systems, explicit control over thread synchronization and context switching overheads is crucial as well. With this in mind, we came to the conclusion that most REEF Drivers and Activities would be best served by an event-driven system.[2]

We considered a number of event driven systems, but found that they were missing crucial features:

Extensibility to multiple I/O subsystems High performance event-driven networking (such as Netty) and disk I/O libraries exist, but do not support the range of applications we target (including legacy libraries, such as JDBC drivers and HDFS handles). Using them would break composability: each data management service would have to be handled by a different REEF subsystem.

Profiling and bottleneck analysis tools similar to those proposed by SEDA [11]. Such tools are the primary benefit of event-driven programming, as they are able to produce per-event handler tracking of throughput, mean event handler latencies, queue lengths, parallelism, and latency histograms [12]. Such information aids debugging and runtime schedulers.

Inlining of event handlers Although Wake's event handling API is purely asynchronous, we often schedule handler invocations in a synchronous fashion by configuring Wake to directly invoke the next handler instead of adding the event to a dispatch queue. This allows aggressive inlining of chains of event handlers [13].

Immutable dataflows Wake event handlers are wired up by invoking object constructors, ensuring that event processing streams are correctly constructed. This avoids complex set-up/tear-down logic in event handlers. It also eliminates a broad class of runtime errors, and associated error handling and runtime overheads. It simplifies Drivers, which can safely assume full knowledge of the dataflows they create (including ones that run remotely, or are written in a different language than the Driver).

7. TANG: JOB CONFIGURATION

REEF Drivers, Evaluators, Activities, and Services are simply implementations of like-named Java or .NET[3] interfaces. Each Job consists of a:

YARN application master that hosts a Java or CLR VM that, in turn, contains a single Driver object, and a

Set of YARN containers that each contain an Evaluator object.

Evaluators, in turn, instantiate Services, and run one Activity at a time until they are shut down. As we implemented REEF, we realized that, in order to compose Activities from multiple frameworks, we needed a way to write general purpose (runtime- and application-independent) code to instantiate these objects.

Early REEF prototypes used ad-hoc logic to process configuration data from properties files (as Hadoop does [14]). Such logic consumed a significant fraction of our code (as in Hadoop), and made hidden assumptions about applications and runtimes. There was no way to move classes between environments, and extending REEF involved mastering obscure design patterns and modifying multiple files. Error handling was a nightmare: there was no distinction between configuration and runtime errors, and no reproducible way to run faulty components in isolation.

To address these issues, we implemented Tang, a configuration manager and dependency injector. Now, REEF extensions consist

[2] Note that here, we focus on the implementations of computational frameworks, not high-level user code.

[3] We plan to add support for additional languages over time.

of standard Java and C# classes and a few annotations. The following properties distinguish Tang from existing approaches:

Static checking of configurations allows jobs to fail earlier and partitions runtime errors into classes: incomplete or wrongly-typed configurations, initialization failures, and runtime errors.

Language independence Drivers written in one language can configure Activities written in multiple languages. This allows REEF to interoperate with the .NET and SQL Server ecosystems, Java-centric DBMS's, and big data solutions built atop Hadoop.

Separation of concerns between configuration processing and Driver implementations. Developers can introduce new types of Drivers and Activities without modifying Tang. In contrast, many dependency injectors are coupled to one application domain (often web services).

Graceful handling of ambiguity for configurations that require further specialization, which enables runtime selection of efficient execution plans.

In short, Tang reduces the problem of bootstrapping a REEF Job to that of shipping configuration files, binaries and any other dependencies to the appropriate Evaluator. Tang automatically checks for and reports the vast majority of application configuration errors and provides detailed diagnostic information when a buggy Driver (or other application code) generates a faulty configuration or makes an unserviceable injection request.

Tang exposes a declarative, dependency-based configuration language that provides the lowest level of composability and (because it is deterministic) fault tolerance in REEF.

8. DEMO DESCRIPTION

We plan to present a REEF cluster running a hybrid MapReduce and machine learning workload. Depending on the job configuration, MapReduce output will be stored on disk or pipelined directly into a machine learning training job. The iterative training algorithm will be capable of making use of per-evaluator retainable state, or of repeatedly scanning over the input data on disk. In addition to this hybrid job, we plan to demonstrate an implementation of page rank based on bulk-synchronous graph processing and an example of REEF's checkpointing mechanisms.

We will also include demonstrations of Tang and Wake. Tang will provide debugging information for misconfigured jobs, perform build-time checks for a range of common mistakes, and emit the information needed to reproducibly run jobs, including the configurations shipped to each Evaluator as the job progresses. For Wake, we will demonstrate profiling tools that pinpoint bottlenecks, measure the effective concurrency achieved by Activities and present request latency histograms from the Driver.

9. CONCLUSION

We plan to use REEF to support a wide range of scalable, fault tolerant computational frameworks. In this demo proposal, we have outlined the primitives that REEF exposes, and explained how they allow many computational frameworks to run atop Hadoop's new resource manager, YARN. We also described Wake, a state of the art event-driven programming framework, and Tang, a configuration manager and dependency injector. Together with the data management primitives that ship with REEF, these systems greatly simplify the implementations of new scalable computational frameworks, and provide compatibility with existing data management infrastructure.

10. REFERENCES

[1] Ananthanarayanan, Ganesh, Ghodsi, Ali, Wang, Andrew, Borthakur, Dhruba, Kandula, Srikanth, Shenker, Scott, and Stoica, Ion. PACMan: Coordinated memory caching for parallel jobs. In *USENIX NSDI* (2012).

[2] Alexandrov, Alexander, Ewen, Stephan, Heimel, Max et al. MapReduce and PACT-comparing data parallel programming models. In *Proceedings of the Conference Datenbanksysteme in Buro, Technik und Wissenschaft (BTW), GI, Bonn, Germany* (2011), 25-44.

[3] Isard, Michael, Budiu, Mihai, Yu, Yuan, Birrell, Andrew, and Fetterly, Dennis. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41, 3 (2007), 59-72.

[4] Borkar, Vinayak, Carey, Michael, Grover, Raman, Onose, Nicola, and Vernica, Rares. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), 1151-1162.

[5] Chu, Cheng, Kim, Sang Kyun, Lin, Yi-An, Yu, YuanYuan, Bradski, Gary, Ng, Andrew Y, and Olukotun, Kunle. Map-reduce for machine learning on multicore. *NIPS*, 19 (2007), 281.

[6] Blumofe, Robert D, Joerg, Christopher F, Kuszmaul, Bradley C, Leiserson, Charles E, Randall, Keith H, and Zhou, Yuli. *Cilk: An efficient multithreaded runtime system*. ACM, 1995.

[7] Hindman, Benjamin, Konwinski, Andy, Zaharia, Matei et al. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI* (2011), 22-22.

[8] Thain, Douglas, Tannenbaum, Todd, and Livny, Miron. Condor and the Grid. *Grid computing: Making the global infrastructure a reality* (2003), 299-335.

[9] Ananthanarayanan, Ganesh, Douglas, Christopher, Ramakrishnan, Raghu, Rao, Sriram, and Stoica, Ion. True elasticity in multi-tenant data-intensive compute clusters. In *ACM SoCC* (2012), 24.

[10] Rao, Sriram, Ramakrishnan, Raghu, Silberstein, Adam, Ovsiannikov, Mike, and Reeves, Damian. Sailfish: A framework for large scale data processing. In *ACM SOCC* (2012), 4.

[11] Welsh, Matt, Culler, David, and Brewer, Eric. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review* (2001), 230-243.

[12] Joukov, Nikolai, Traeger, Avishay, Iyer, Rakesh, Wright, Charles P, and Zadok, Erez. Operating system profiling via latency analysis. In *OSDI* (2006), 89-102.

[13] Kohler, Eddie, Morris, Robert, Chen, Benjie, Jannotti, John, and Kaashoek, M Frans. The Click modular router. *ACM TOCS*, 18, 3 (2000), 263-297.

[14] Rabkin, Ariel. *Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software*. Dissertation, UC Berkeley, 2012.