Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs

Lu Fang[1] Khanh Nguyen[1] Guoqing Xu[1] Brian Demsky[1] Shan Lu[2]

University of California, Irvine[1] University of Chicago[2]

{lfang3, khanhtn1, guoqingx, bdemsky}@uci.edu shanlu@uchicago.edu

Abstract

Real-world data-parallel programs commonly suffer from great *memory pressure*, especially when they are executed to process large datasets. Memory problems lead to excessive GC effort and out-of-memory errors, significantly hurting system performance and scalability. This paper proposes a systematic approach that can help data-parallel tasks *survive* memory pressure, improving their performance and scalability without needing any manual effort to tune system parameters. Our approach advocates *interruptible task* (ITask), a new type of data-parallel tasks that can be interrupted upon memory pressure—with part or all of their used memory reclaimed—and resumed when the pressure goes away.

To support ITasks, we propose a novel programming model and a runtime system, and have instantiated them on two state-of-the-art platforms *Hadoop* and *Hyracks*. A thorough evaluation demonstrates the effectiveness of ITask: it has helped real-world Hadoop programs survive 13 out-of-memory problems reported on StackOverflow; a second set of experiments with 5 already well-tuned programs in Hyracks on datasets of different sizes shows that the ITask-based versions are 1.5–3× faster and scale to 3–24× larger datasets than their regular counterparts.

1. Introduction

A key challenge in grappling with the explosion of Big Data is to develop *scalable* software systems that can efficiently process massive amounts of data. Although much work has been done to improve scalability at the architecture level for distributed systems [19, 30–33, 41–43, 48], a common problem in practice is memory pressure [26, 47] on individual nodes—the execution pushes the heap limit soon after it starts and the system struggles to find memory

SOSP '15, October 4–7, 2015, Monterey, CA, USA.

 ISBN 978-1-4503-3834-9/15/10. . . $15.00.

http://dx.doi.org/10.1145/2815400.2815407

to allocate new objects throughout the execution. These problems are further exacerbated by the pervasive use of managed languages such as Java and C#, which often blow up memory usage in unpredictable ways [44, 53, 54]. Large memory pressure leads to not only poor performance (e.g., with more than 50% of time spent in garbage collection [26]), but also execution failures due to out-of-memory errors (OMEs).

Motivation No existing technique can systematically address the individual node memory pressure problem. Practical solutions center around making "best practice" recommendations for manual tuning of framework parameters [18]. For example, the developer could reduce input size for each task (i.e., finer granularity) and/or degree-of-parallelism (i.e., fewer threads). However, it is impossible to find a one-size-fits-all configuration even for a single data-parallel program when considering data skewness and different behaviors of its tasks, not to mention finding an optimal configuration across different programs.

In a real example [17], the program analyzes all posts on StackOverflow. There exists a small number of popular posts with extremely long discussion threads. Memory pressure occurs when such a long post is processed in a Reducer in Hadoop: the execution of one single thread can use up almost the entire heap. If another thread is running simultaneously to process other posts, the program is bound to run out of memory. This problem can be solved by changing the framework parameter to always run one single-threaded task instance on each node. However, shorter posts can be processed perfectly in parallel; making the entire framework sequential to handle few long posts is clearly an overkill.

Since manual tuning is difficult and requires highly-specialized expertise, much work has been done to develop automated tuning tools, including efforts from both industry (e.g., YARN [3]) and academia (e.g., Mesos [37] and Starfish [36]). These intelligent schedulers allocate resources by predicting a task's future resource usage based on its past utilization. However, memory behaviors are very difficult to predict. In the example discussed above, YARN schedules a task to process a long post on a node where other tasks are already running, based on the observation that a "normal" task did not take much memory in the past. Precise prediction of memory behaviors is almost impossible when considering the wide variety of datasets and the rich semantics of tasks.

Many frameworks now support *out-of-core* computations (such as spilling in Hadoop and Spark or dumping stale objects to disk [23, 50]) to reduce memory pressure. However, those out-of-core algorithms are framework-specific and not designed to help general data-parallel tasks survive when they are about to run out of memory.

Our contributions We propose a novel, systematic approach, called *interruptible task (ITask)*, that can help data-parallel programs survive memory pressure without needing (1) additional hardware resources (e.g., memory, nodes, etc.) or (2) manual parameter tuning. Inspired by how processors handle hardware interrupts, our idea is to *treat memory pressure as interrupts*: a data-parallel task can be interrupted upon memory pressure—with part or all of its consumed memory reclaimed—and re-activated when the pressure goes away.

ITask provides the following unique features unseen in existing systems. First, ITask works *proactively* in response to memory pressure. We take actions to interrupt tasks and reclaim memory when we observe the first signs of pressure. Hence, ITask can quickly take the system back to the memory "safe zone" before much time is spent on garbage collection (GC) and way before an OME occurs. As a result, ITask improves both scalability (because out-of-memory crashes are avoided) and performance (because GC time is reduced).

Second, ITask uses a *staged approach* to lower memory consumption for an interrupted task t. It consists of five steps, covering all components of a running task's user-level memory

usage with varying cost-benefit tradeoffs: (i) heap objects referenced by local variables during t's execution are all released; (ii) the part of the input data already processed by t is released; (iii) final results generated by t are pushed out; (iv) intermediate results that need to be aggregated before being pushed out will be aggregated by a follow-up task in an out-of-core manner; and (v) other in-memory data are serialized (e.g., to disk). Not all of these steps will be performed at every interrupt: the handling is done *lazily* and it stops whenever memory pressure disappears.

Third, ITask consists of a new programming model and a runtime system that can be easily implemented in any existing data-parallel framework. The programming model provides interrupt handling abstractions for developers to reason about interrupts. The ITask runtime system (IRS) performs task scheduling and runtime adaptation. The amount of work needed to implement ITask is minimal: the user simply restructures code written for existing data-parallel tasks to follow new interfaces, and the framework needs to be slightly modified to delegate task scheduling to the IRS. The IRS sits *on top of* the framework's job scheduler, providing complementary optimizations and safety guarantees.

Summary of results We have instantiated ITasks in the widely-used Hadoop framework [21] as well as Hyracks [24], a distributed data-parallel framework. We reproduced 13 Hadoop problems reported on StackOverflow and implemented their ITask versions. ITask was able to help all of these programs survive memory pressure and successfully process their entire datasets. For a diverse array of 5 problems, we performed an additional comparison between their ITask versions under default configurations and their original programs under the recommended configurations; the results show that the ITask versions outperform the manually-tuned versions by an average of 2×.

For Hyracks, we selected 5 well-tuned programs from its repository and performed a detailed study on performance and scalability between their original versions and ITask versions: the ITask versions are 1.5–3× faster and scale to 3–24× larger datasets than their regular counterparts. In fact, the scalability improvement ITask provides can be even higher (e.g., hundreds of times) if further larger datasets are used.

2. Memory Problems in the Real World

To understand memory problems and their root causes in real-world data-parallel systems, we searched in StackOverflow for the two keywords "out of memory" and "data parallel". The search returned 126 memory problems, from which we discarded those that can be easily fixed or whose problem description was not sufficiently clear for us to reproduce. We eventually obtained 73 relevant posts with clear problem descriptions. A detailed investigation of them identified two common root-cause patterns:

Hot keys In a typical data-parallel framework, data are represented as key-value pairs, and some particular keys may have large numbers of associated values. 22 problems fall into this category, including the example discussed in §1—a join operation in the program builds an XML object that represents a post (with all its comments) on StackOverflow. Certain posts are much longer than others; holding one such long and popular post can consume an extremely large amount of memory. Consequently, the OME can only be avoided if the processing of a long post does not occur simultaneously with that of other posts, which, in an existing framework, can only be done by making the entire framework sequential.

Large intermediate results For 51 problems, the semantics of their programs require intermediate results to be cached in memory, waiting for subsequent operations before the final results can be produced. In many cases, developers hold these results in large Java

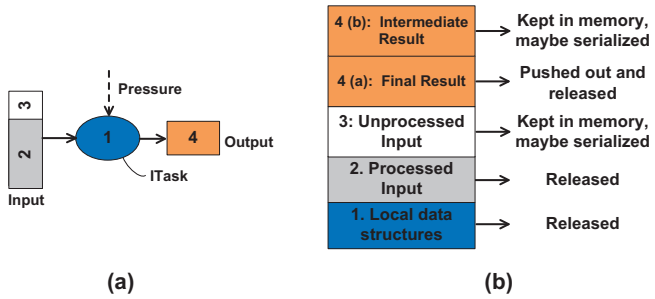


Figure 1. A graphical illustration of (a) an ITask execution at an interrupt with numbers showing different components of its memory consumption; and (b) how these components are handled.

collections (e.g., HashMap or ArrayList), which have non-trivial space overhead [45]. One post [10] describes a scenario in which the developer uses the Stanford Lemmatizer (i.e., part of a natural language processor) to preprocess customer reviews before calculating the lemmas' statistics. The task fails to preprocess a dataset of 31GB at a very early stage. The large memory consumption of the lemmatizer is the cause: due to the temporary data structures used for dynamic programming, for each sentence processed, the amount of memory needed by the lemmatizer is 3 orders of magnitude larger than the sentence. Furthermore, the Stanford Lemmatizer is a third-party library: the developer is unlikely to know either its memory behavior or how to control it.

Among the 73 posts we studied, only 25 have recommended fixes. We also investigated these recommendations and classified them into two major categories:

Configuration tuning There are 16 problems for which the recommended fixes are to change framework parameters. However, framework parameters in data-parallel systems are extremely complex. For example, Hadoop has about 190 framework parameters, such as data split size, number of workers, buffer size, etc. Even experienced developers may have difficulties finding the appropriate configurations. The tuning process is often labor-intensive, consisting of repetitive tests and trials. In fact, almost every discussion thread contains multiple proposed parameter changes and there is no confirmation whether they have actually worked.

Skew fixing There are 9 posts in which the recommended fixes are to fix skews in the datasets. For instance, to fix the lemmatizer problem, one recommended a thorough profiling to find all long sentences in the dataset and break them into short sentences. However, it is nearly impossible to manually break sentences in such a large dataset. While there exist tools that can manage skews [40], it is difficult to apply them to general datasets, where there is often huge diversity in data types and distributions.

The complexity of real-world memory problems as well as the difficulty of manually coming up with fixes strongly call for system support that can automatically release memory upon extreme memory pressure. The design of ITask is motivated exactly by this real-world need. We have implemented ITask versions for a representative subset of the problems we have studied. Without any manual parameter tuning or skew fixing, the ITask versions successfully processed the datasets on which their original versions crashed with OMEs.

3. Design Overview

The high level idea behind ITasks is shown in Figure 3. When the system detects memory pressure, a selected task is interrupted, with part or all of its consumed memory reclaimed. This process is repeated until the pressure disappears to direct the execution of other tasks on the same node back to the "safe zone" of memory usage.

We present below three key challenges in carrying out this high-level idea, our high-level solutions to these challenges, as well as the ITask system architecture.

How to lower memory usage when a task is interrupted? As shown in Figure 1 (a) and (b), the memory consumption of a data-parallel task instance consists of the following four components: (1) local data structures created by the task, (2) the processed input data before the interrupt, (3) the unprocessed part of the input, and (4) the partial results produced. Simply blocking a thread running the task without swapping data would not change the task's memory consumption at all; naïvely terminating the thread can completely eliminate the task's memory consumption, but would also completely waste the computation already performed by the thread.

Our design carefully handles different memory components differently when terminating the task-running thread, as shown in Figure 1. For Component 1 and Component 2, it is safe to discard them when the corresponding thread is terminated. For Component 3, we will try to keep the unprocessed input in memory and serialize it (lazily) when needed.

For Component 4, we differentiate two sub-types of result data, represented by 4(a) and 4(b) in Figure 1. Immediately useful results, referred to as *final results*, can be pushed to the next operator in the data pipeline immediately (e.g., another set of MapReduce tasks). Where that next operator is executed is determined by the framework scheduler and may be on a different node. Results that are not immediately useful and need further aggregation, referred to as *intermediate results*, will stay in memory and wait to be aggregated until all intermediate results for the same input are produced. These results can be lazily serialized under severe memory pressure. Which result is final and which is intermediate depends on the task semantics. For example, in MapReduce, an interrupt to a Map task generates a final result, which can be forwarded immediately to the shuffle phase; an interrupt to a Reduce task generates an intermediate result, which cannot be used until all intermediate results from the same hash bucket are aggregated.

When to interrupt a task? The best timing has to consider two factors: per-process system memory availability and per-thread/task data processing status. Specifically, we want to interrupt a task when the overall memory pressure comes and when its execution arrives at a *safe state* where it is not in the middle of processing an input data item. The former avoids unnecessary interrupts. The latter allows terminating a task by recording only minimum local information of the execution. During task re-activation, a new task instance can simply work on the unprocessed part of the original input without missing a beat.

To handle the first factor, our system leverages an observation that long and useless GC (LUGC)—that scans the whole heap without reclaiming much memory—is a good indicator of memory pressure, and uses an LUGC as a signal to trigger interrupts. To handle the second factor, we need to understand the data processing status, which is related to the semantics of a task.

How to interrupt a task? Interrupting in our system involves much more than terminating a random thread in a memory-pressured process. In a system with many tasks running, determining which thread(s) to terminate is challenging and requires precise global runtime assessment. Even if we know which task to interrupt, conducting the interrupt is still non-

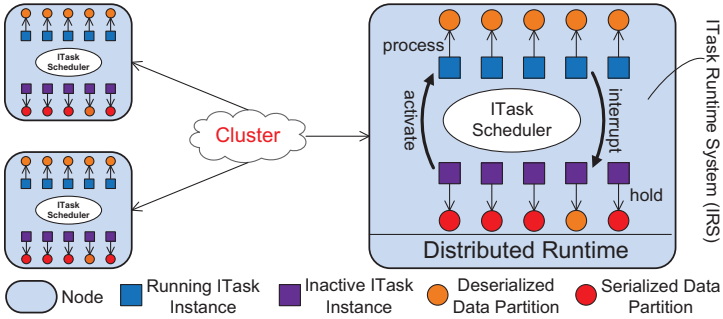


Figure 2. The architecture of the ITask Runtime System.

trivial, involving recording the progress of the task, such as which part of the input has been processed and what results have been generated. Like the two challenges stated above, addressing this challenge requires both whole-system coordination and understanding of per-task semantics.

ITask system architecture The ITask system includes two main components that work together to address the challenges discussed above: an ITask programming model and an ITask runtime system (IRS).

The ITask programming model provides abstractions for developers to program interrupt logic and allows the IRS to conduct operations that require semantic information about tasks. Our programming model is carefully designed to provide a *non-intrusive* way to realize ITasks, making it possible to quickly modify existing data-parallel programs and enable ITasks in different frameworks. Specifically, developers only need to implement a few additional functions by restructuring code from an existing task. The details of the programming model are discussed in §4.

The IRS is implemented as a Java library. This library-based implementation makes ITask immediately applicable to all existing Java-based data-parallel frameworks. The same idea can be easily applied to other managed languages such as C# and Scala. More details of the IRS implementation are presented in §5.

Figure 2 shows a snapshot of a running ITask-based data-parallel job. The IRS sits on top of the distributed runtime on each node. Whenever the job scheduler assigns a job, which contains a set of ITasks implementing a logical functionality, to a machine, it *submits* the job to the IRS instead of directly running the ITasks. The IRS maintains a task scheduler that decides when to interrupt or re-activate an ITask instance. As a result, the number of running ITask instances dynamically fluctuates in response to the system's memory availability. Each task instance is associated with an input data partition. Running tasks have their inputs in the deserialized form in memory (e.g., map, list, etc.), while interrupted tasks may have their inputs in the serialized form (e.g., bytes in memory or on disk) to reduce memory/GC costs.

Figure 3 illustrates an over-the-time memory-usage comparison between executions with and without ITasks. In a normal execution, the memory footprint keeps increasing; after a few LUGCs, the program crashes with an OME. In an ITask execution, the IRS starts interrupting tasks at the first LUGC point; the memory usage is brought down by the ITask interrupt and stays mostly constant until the next re-activation point at which new task instances are created. Without any manual configuration tuning, ITask-enhanced data-parallel jobs can keep their memory consumption in the safe zone throughout their executions, effectively avoiding wasteful GC effort and disruptive OMEs.

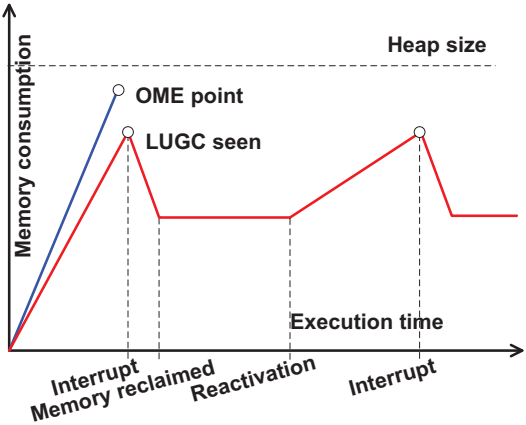


Figure 3. A high-level memory footprint comparison between executions with (in red) and without (in blue) ITasks.

Other design choices Besides the proposed approach, there are several other design choices. The first one is to develop a language with new constructs to allow developers to express interrupt logic. However, as with all language design efforts, there is an inherent risk that developers lack strong motivation to learn and use the new language. Another choice is to develop a data-parallel system from scratch with all the ITask-related features embedded. This choice shares a similar risk to language design: migrating programs from existing frameworks to a new framework is a daunting task that developers would be reluctant to do. Hence, it is clear to us that the most practical way to systematically reduce memory pressure is to combine an API-based programming model with a library-based runtime system—as proposed in the paper—that can extend the performance benefit to a large number of existing systems and programs independently of their computation models.

4. The ITask Programming Model

This section describes the ITask programming model as well as how it is implemented in two state-of-the-art data-parallel frameworks: Hyracks [24] and Hadoop [21].

4.1 Programming Model

To turn an existing data-parallel task into an ITask, the developer needs to make the following three changes to the task's original implementation. First, implement the `DataPartition` interface (shown in the upper part of Figure 4). `DataPartition` objects wrap around the framework's existing data representation (e.g., key-value buffer) and are used as an ITask's input and output. Second, make the original task's Java class inherit the `ITask` abstract class (shown in the lower part of Figure 4) and implement its four abstract methods. This can be easily done by restructuring existing code and adding a small amount of new code to handle interrupts. Third, add a few lines of glue code to specify the input-output relationships between data partitions and ITasks. The development effort is insignificant because most of the work is moving existing code into different methods. For instance, for the 13 StackOverflow problems we reproduced, it took us only a week in total to implement their ITask versions although we had never studied these programs before.

Input and output For a data-parallel task, the input dataset is a vector of *data tuples*. A `DataPartition` object wraps around an interval of tuples in the input and different

```
1  // The DataPartition abstract class in the library
2  abstract class DataPartition {
3    int tag, cursor; // Partition state
4    abstract boolean hasNext();
5    abstract Tuple next();
6    abstract void serialize();
7    abstract DataPartition deserialize();
8  }
9  // The ITask abstract class in the library
10 abstract class ITask {
11   // Initialization logic
12   abstract void initialize();
13   // Interrupt logic
14   abstract void interrupt();
15   // Finalization logic
16   abstract void cleanup();
17   // Process a tuple; this method should be side-effect-free
18   abstract void process(Tuple t);
19   // Scalable loop
20   boolean scaleLoop(DataPartition dp) {
21     initialize();
22     while (dp.hasNext()) {
23       if (Monitor.hasMemoryPressure() &&
24           ITaskScheduler.terminate(this)) {
25         // Invoke the user-defined interrupt logic
26         interrupt();
27         // Push the partially processed input to the queue
28         ITaskScheduler.pushToQueue(dp);
29         return false;
30       }
31       process(dp.next());
32     }
33     cleanup();
34     return true;
35   }
36 }
```

Figure 4. The `DataPartition` and `ITask` abstract classes; abstract methods that the user needs to implement are highlighted in red.

partitions never overlap. Existing data-parallel frameworks already have their own notions of partitions; to use ITask, the developer only needs to wrap an existing partition in a `DataPartition` object.

The `DataPartition` class provides a unified interface for the runtime to track the state of data processing. The internal state of a partition object (Line 3) has two major components: (1) a *tag* that specifies how partial results should be aggregated, and (2) a *cursor* that marks the boundary between the processed and unprocessed parts of the input. We will discuss these two components shortly.

The `DataPartition` class also provides an interface to iterate over data items through the `next` and `hasNext` methods as well as `serialize` and `deserialize` methods to convert data format. It is up to the developer how to implement `serialize` and `deserialize`: the data partition can be serialized to disk and the deserialization brings it back into memory; for applications that cannot tolerate disk I/O, the partition can be serialized to large byte arrays and the deserialization recovers the object-based representation. These two methods will be invoked by the IRS to (de)serialize data in response to memory availability (see §5).

The ITask abstract class An existing task needs to extend the `ITask` abstract class to become an interruptible task, as shown in Figure 4. The extension forces the task to implement four new methods: `initialize`, `interrupt`, `cleanup`, and `process`. As we will show shortly, the implementation can be easily done by simply re-structuring existing

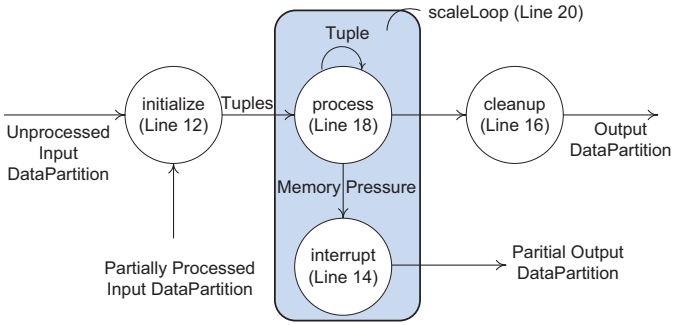


Figure 5. The data flow of an ITask execution.

code. Rather than changing the original core semantics of a task (e.g., map or reduce in Hadoop), these methods provide a way for the developer to reason about interrupts.

The `initialize` method loads the input and creates local (auxiliary) data structures before starting data processing; `interrupt` specifies the interrupt handling logic; `cleanup` contains the finalization logic when the *entire* input is processed; and `process` implements the main data processing logic. The `scaleLoop` method is implemented in the library. It iterates over the input data tuples and invokes the `process` method to process a tuple in each iteration (Line 31). It checks memory availability at the beginning of each iteration, ensuring that interrupts can only occur at safe points (i.e., not in the middle of processing a tuple).

When a running task is interrupted, its input and output data partitions still stay in memory unless explicitly released by the developer (such as Line 12 in Figure 6). Serialization is not immediately performed. Instead, it is done in a *lazy* manner by the partition manager (§5.3) when needed.

Input-output relationship The invocation of an ITask has a *dataflow semantics*, which aligns with the dataflow nature of the framework: as long as (1) there exists a `DataPartition` object in the partition queue, which contains the inputs of all tasks, and (2) the cursor of the partition does not point to the end, the task will be automatically invoked to process this partition. To illustrate, consider the following code snippet,

```
1 ITaskA.setInputType(DataPartitionA.class);
2 ITaskA.setOutputType(DataPartitionB.class);
3 ITaskB.setInputType(DataPartitionB.class);
4 ITaskB.setOutputType(DataPartitionC.class);
```

in which `ITaskA` and `ITaskB` are two `ITask` classes; and `DataPartitionA` and `DataPartitionB` are two `DataPartition` classes. These four statements make `ITaskB` a successor of `ITaskA`: whenever a `DataPartitionB` is produced by `ITaskA`, it can be immediately processed by `ITaskB`.

ITask State Machine Putting it all together, Figure 5 shows the ITask state machine. The input of the ITask is a `DataPartition` object in the partition queue (see §5.3). The object represents either a new, unprocessed partition or a partially processed partition that was pushed into the queue at a previous interrupt. As shown in Figure 4, after initialization (Line 12), the data is forwarded to `scaleLoop` (Line 20), which invokes the user-defined `process` (Line 18) method to process tuples. Each iteration of the loop processes one tuple and increments the input cursor. If there are no more data items to process (i.e., the cursor

points to the end of the partition), the execution proceeds to `cleanup` (Line 16), which releases resources and outputs a new `DataPartition` object. This partition will become the input of the next ITask in the pipeline.

Upon memory pressure (Line 23), if the IRS determines that the current ITask instance needs to be terminated (Line 24, based on a set of priority rules discussed in §5), the user-defined interrupt logic is executed (Line 14) and the input data partition is pushed into the partition queue. The `scaleLoop` is then exited, and this terminates the thread and produces an output partition representing a result. The input partition cursor marks the boundary between processed and unprocessed tuples. Future processing of this partition will start at the cursor when memory becomes available.

An important requirement here is that the `process` method cannot have side effects: it is only allowed to write the output partition and internal objects, which guarantees that the processing of a partially-processed data partition can be resumed without needing to restore a particular external state. Note that this is not a new requirement since side-effect freedom has already been enforced in many existing data-parallel tasks, such as Map/Reduce in Hadoop.

ITask with multiple inputs It is necessary to allow an ITask to process multiple inputs at the same time, especially when a task produces an intermediate result that cannot be directly fed to the next task in the pipeline, as shown by component 4 (b) in Figure 1. As discussed earlier, an interrupt to a Reduce task in MapReduce would produce such an intermediate result; an additional follow-up task is needed to aggregate all intermediate results before a final result can be produced and further processed. To enable aggregation, we design an abstract class called `MITask`, which takes multiple data partitions as input. This class differs from `ITask` only in the definition of the `scaleLoop` method:

```
1  abstract class MITask {
2    boolean scaleLoop(PartitionIterator<DataPartition> i) {
3      initialize();
4      while (i.hasNext()) {
5        DataPartition dp = (DataPartition) i.next();
6        while (dp.hasNext()) {
7          if (...) { // The same memory availability check
8            interrupt();
9            ITaskScheduler.pushToQueue(i);
10           return false;
11         }
12         process(dp.next());
13       } }
14     cleanup();
15     return true;
16   } ... }
```

In `MITask`, `scaleLoop` processes a set of `DataPartition` objects. Since the partition queue may have many partition objects available, a challenge here is how to select partitions to invoke an MITask. We overcome the challenge using tags (Line 3 in Figure 4): each instance (thread) of an MITask is created to process a set of `DataPartition` objects that have the same tag. Tagging should be done in an earlier ITask that produces these partitions. Obviously, multiple MITask instances can be launched in parallel to process different groups of data partitions with distinct tags. Note that a special iterator `PartitionIterator` is used to iterate over partitions. It is a *lazy, out-of-core* iterator that does not need all partitions to be simultaneously present in memory; a partition on disk is loaded only if it is about to be visited.

```
1  // The Map ITask for Hyracks
2  class MapOperator extends ITask
3                    implements HyracksOperator {
4    MapPartition output;
5    void initialize() {
6      // Create output partition
7      output = new MapPartition();
8    }
9    void interrupt() {
10     // The output can be sent to shuffling at any time
11     Hyracks.pushToShuffle(output.getData());
12     PartitionManager.release(output);
13   }
14   void cleanup() {
15     Hyracks.pushToShuffle(output.getData());
16   }
17   void process(Tuple t) {
18     addWordInMap(output, t.getElement(0));
19   }
20   // A method defined in HyracksOperator
21   void nextFrame(ByteBuffer frame) {
22     // Wrap the buffer into a partition object
23     BufferPartition b = new BufferPartition(frame);
24     // Set input and output
25     MapOperator.setInputType(BufferPartition.class);
26     MapOperator.setOutputType(MapPartition.class);
27     // Push the partition to the queue and run the ITask
28     ITaskScheduler.pushToQueue(b);
29     ITaskScheduler.start();
30   }
31 }
```

Figure 6. The ITask implementation of the `MapOperator` for the `WordCount` application in Hyracks. (Hyracks classes/methods are in green.)

4.2 Instantiating ITasks in Existing Frameworks

Hyracks Hyracks [4] is a carefully crafted distributed dataflow system, which has been shown to outperform Hadoop and Mahout for many workloads [24]. In Hyracks, the user specifies a dataflow graph in which each node represents a data operator and each edge represents a connection between two operators. Processing a dataset is done by pushing the data along the edges of the graph. For example, given a big text file for `WordCount` (WC), Hyracks splits it into a set of (disjoint) partitions, each of which will be assigned to a worker for processing. Hyracks users dictate the data processing by implementing the `MapOperator` and `ReduceOperator`, connected by a "hashing" connector. The main entry of each operator's processing is the `nextFrame` method.

The `Map` operator does local word counting in each data partition. Its ITask implementation is shown in Figure 6. To launch ITasks from Hyracks, we only need to write five lines of code as shown in Lines 21–30 in Figure 6: in the `nextFrame` method, we create a `BufferPartition` object using the `ByteBuffer` provided by the framework, set the input-output relationship, and start the ITask execution engine. The `nextFrame` method will be invoked multiple times by the Hyracks framework, and hence, multiple `BufferPartition` objects will be processed by threads running `MapOperator` under the control of the IRS. Upon memory pressure, the `MapPartition` object, *output*, contains a *final result*, and thus the invocation of `interrupt` can directly send it to the shuffle phase (Line 11).

The `Reduce` operator re-counts words that belong to the same hash bucket. Figure 7 shows its ITask implementation. `Reduce` has a similar implementation to `Map`. However, when memory pressure occurs and a `Reduce` thread is terminated, *output* contains *intermediate*

```
1  // The Reduce ITask for Hyracks
2  class ReduceOperator extends ITask
3                       implements HyracksOperator {
4    MapPartition output;
5    void initialize() {
6      // Create output partition
7      output = new MapPartition();
8    }
9    void interrupt() {
10     // Tag the output with the ID of hash bucket
11     output.setTag(Hyracks.getChannelID());
12     ITaskScheduler.pushToQueue(output);
13   }
14   void cleanup() {
15     output.setTag(Hyracks.getChannelID());
16     ITaskScheduler.pushToQueue(output);
17   }
18   void process(Tuple t) {
19     addWordInMap(output, t.getElement(0));
20   }
21   void nextFrame(ByteBuffer frame) {
22     BufferPartition b = new BufferPartition(frame);
23     // Connect ReduceOperator with MergeTask
24     ReduceOperator.setInputType(BufferPartition.class);
25     ReduceOperator.setOutputType(MapPartition.class);
26     MergeTask.setInputType(MapPartition.class);
27     ITaskScheduler.pushToQueue(b);
28     ITaskScheduler.start();
29   }
30 }
31 // The Merge MITask for Hyracks
32 class MergeTask extends MITask {
33   MapPartition output;
34   void initialize() {
35     // Create output partition
36     output = new MapPartition();
37   }
38   void interrupt() {
39     output.setTag(input.getTag());
40     ITaskScheduler.pushToQueue(output);
41   }
42   void cleanup() {
43     Hyracks.outputToHDFS(output);
44   }
45   void process(Tuple t) {
46     aggregateCount(output, t.element(0), t.element(1));
47   }
48 }
```

Figure 7. The ITask implementation of the `ReduceOperator` for the `WordCount` application in Hyracks.

results that are not immediately useful. Hence, before the thread is terminated, we tag *output* with the ID of the channel that the input `ByteBuffer` comes from (Lines 11, 15). Because a distinct channel is used for each hash bucket, tagging facilitates the future recognition of partial results that belong to the same hash bucket.

Next, we develop an MITask `MergeTask` that aggregates the intermediate results produced by the interrupted `Reduce` instances. If a `MergeTask` instance is interrupted, it would create further intermediate results; since these results are tagged with the same tag as their input (Line 39), they will become inputs to `MergeTask` itself when memory becomes available in the future.

The careful reader may notice that `MergeTask` has a similar flavor to the merge phase proposed in the Map-Reduce-Merge model [56]. Here the merge task is simply an example

of the more general MITask that does not have a specific semantics while the merge phase in [56] has a fixed semantics to merge the reduce results. MITask can be instantiated to implement any M-to-N connector between different data partitions. In this work, an MITask assumes *associativity* and *commutativity* among its input partitions (with the same tag). Because these partitions can be accessed in an arbitrary order, the MITask may compute wrong results if some kind of ordering exists among them. Note that this is a natural requirement for any data-parallel task—any ordering may create data dependencies, making it difficult to decompose the input and parallelize the processing.

Note that the code that achieves the core functionality of the two operators already exists in the original implementations of their `nextFrame` method. Turning the two operators to ITasks requires only lightweight code restructuring.

Hadoop Hadoop is a MapReduce framework in which tasks only need to extend the `Mapper`/`Reducer` abstract classes, and therefore have narrower semantics than Hyracks tasks. To enable ITask in Hadoop, we let `Mapper` and `Reducer` extend `ITask`, so that all user-defined tasks automatically become ITasks. In addition, the `run` method in `Mapper`/`Reducer` is modified to become a driver to invoke the ITask state machine; its original functionality is moved into the `scaleLoop` method, as shown in the following code snippet of Mapper.

```
1  class Mapper extends ITask {
2    ... // Implementations of the ITask methods.
3    void run() {
4      initialize();
5      if(!scaleLoop()) return;
6      cleanup();
7    }
8  }
9  class MyMapper extends Mapper {
10   void map(T key, K value, ...) {
11     ... // Data processing logic
12   }
13   void process(Tuple t) {
14     map(t.getElement(0), t.getElement(1));
15   }
16 }
```

Other frameworks It is possible to instantiate ITasks in other data-parallel frameworks as well, such as Spark [60] and Dryad [38]. In general, this can be done by embedding the ITask state machine into the semantics of existing tasks, an easy effort that can be quickly done by experienced programmers. Furthermore, the majority of the IRS code can be reused across frameworks; slight modification is needed only for the *boundary code* where the IRS interacts with the framework.

4.3 Discussion

Through these examples, it is clear to see the necessity of providing abstractions for developers to reason about interrupts. In fact, for the three tasks in the Hyracks WC example, the handling of interrupts is completely different: when a Map thread is interrupted, the output can be directly sent to shuffling; when a Reduce thread is interrupted, its output needs to be tagged with the hash bucket ID so that it can be appropriately processed by the Merge task; when a Merge thread is interrupted, its output also needs to be tagged appropriately so that it will become its own input. Without the ITask programming model, it is difficult to customize interrupt handling based on the task semantics.

Our experience shows that the effort of writing ITasks is small — since the data processing logic already exists, refactoring a regular task into an ITask usually requires the developer to manually write less than 100 lines of code. For example, the ITask version of WC has 309

more lines of code than its regular counterpart. 226 lines, including function skeletons and glue code, can be automatically generated by an IDE or our static analyzer. By default, we employ a third-party library called Kryo [39] to perform data serialization and deserialization; the use of this library only requires a few lines of code for object creation and method calls. The developer may also write their own serialization and deserialization logic to optimize I/O performance.

Many data-parallel tasks are generated from high-level declarative languages. For example, Hyracks hosts the AsterixDB [1] software stack while Hadoop has a large number of query languages built on top of it, such as Hive [51] and Pig Latin [48]. Currently, we rely on developers to manually port existing tasks to ITasks. Once the usefulness of ITasks is demonstrated, an important and promising future direction is to modify the compilers of those high-level languages to make them automatically generate ITask code.

5. The ITasks Runtime System

Once enabled, the IRS manages task scheduling. The IRS contains three components: the partition manager, the scheduler, and the monitor. It determines (1) when to interrupt or re-activate an ITask (monitor, §5.2), (2) when to serialize or deserialize data partitions (partition manager, §5.3) , and (3) which ITask to interrupt or re-activate (scheduler, §5.4).

5.1 The IRS Overview

The IRS starts with a *warm-up* phase in which a slow-start parallelism model is used to gradually scale up the number of threads: initially one thread is created to run the entry task; as the task executes, our system gradually increases the number of threads until it reaches the optimal execution point. If the heap is sufficiently large, the optimal execution point is where the number of threads equals the number of logical cores in the system, which defines the maximum amount of parallelism one can exploit. Otherwise, the IRS stops increasing the number of threads at the moment the available memory size falls below a user-defined threshold percentage (e.g., $N\%$ of the total heap) to guarantee that the program is executed in a pressure-free environment. The warm-up phase serves as an initial guard to managing memory consumption: although the system memory utilization may change later, the IRS is unlikely to need to tune memory usage immediately after the program starts.

From the task code, we develop a static analysis that builds a *task graph* based on the input/output relationship of the ITasks in the program. The task graph will be used later to determine which ITask instances should be interrupted and re-activated. Figure 8 shows a high-level algorithm explaining the interaction among the three IRS components.

5.2 Monitor

The IRS monitors the global memory usage and notifies the scheduler of the system's memory availability. As discussed earlier, we design the monitor by focusing on LUGCs. Specifically, we consider a GC as a LUGC if the GC cannot increase the free memory size above M% of the heap size, where M is a user-specified parameter. The monitor also watches the execution to identify periods in which extra memory is available. These periods occur when the size of free memory is $\geq$ N% of the heap size. If such a period is detected, the monitor sends a "GROW" signal (Line 7) to instruct the scheduler to increase the number of ITask instances. The scheduler then picks a task, finds a partition object that can serve as its input, and creates a new thread to run it (Lines 19–24). We used N = 20 and M = 10 in our experiments and they worked well.

```
1:  /* Monitor */
2:  while true do
3:     if Long and Useless GC occurs then
4:        SIGNALSCHEDULER("REDUCE") // §5.2
5:     end if
6:     if freeHeap ≥ N% ∗ totalHeap then
7:        SIGNALSCHEDULER("GROW") // §5.2
8:     end if
9:  end while
10:
11: /* Scheduler */
12: while Message m = LISTENTOMONITOR() do
13:    if m == "REDUCE" then
14:       SIGNALPARTITIONMANANGER("SERIALIZE")
15:       while freeHeap < M% ∗ totalHeap do
16:          INTERRUPTTASKINSTANCE() // §5.4
17:       end while
18:    end if
19:    if m =="GROW" then
20:       while freeHeap ≥ N% ∗ totalHeap do
21:          dp = PM.FINDAVAILABLEPARTITION()
22:          INCREASETASKINSTANCE(dp) // §5.4
23:       end while
24:    end if
25: end while
26:
27: /* Partition Manager */
28: while Message m = LISTENTOSCHEDULER() do
29:    if m == "SERIALIZE" then
30:       SCANANDDUMP() // §5.3
31:    end if
32: end while
```

Figure 8. The interaction between the monitor, scheduler, and partition manager.

5.3 Partition Manager

Once a partition object is created, such as in the `nextFrame` method in Figure 6, it is registered with the partition manager. The manager puts the object into a global partition queue (as mentioned earlier) that contains all partially-processed and unprocessed partitions. These data partitions may be in serialized or deserialized form. How a partition is serialized depends on the `serialize` method defined in the partition class. While there can be multiple ways to implement the method, in our current prototype, data serialization writes a partition to disk and deserialization brings it back to memory. To avoid thrashing, we keep track of each partition's latest serialization and deserialization timestamps. A data partition is not allowed to be serialized if a deserialization of the partition was performed recently within a given time period, unless there are no other data partitions with earlier deserialization timestamps. If thrashing still occurs, the partition manager notifies the monitor, which then sends a "REDUCE" signal to the scheduler to terminate threads.

Upon receiving a "REDUCE" signal from the monitor, the scheduler first checks with the partition manager to see if it can serialize some data partitions that are associated with already interrupted tasks (Lines 14, 28–32). In many cases, this is sufficient to remove

memory pressure so that we do not need to interrupt more tasks. The partition manager uses the following rules to determine which partitions to serialize first. Background threads then write the data to disk.

- *Temporal Locality Rule:* Partitions that serve as inputs to the ITasks that are closer to the currently executed ITask on the task graph have a higher priority to stay in memory.
- *Finish Line Rule:* A fast turn-around from the initial inputs to the final output is a desired property of any system. To optimize for this property, the inputs to the ITasks that are closer to the finish line (i.e., lower on the task graph) have a higher priority to be retained in memory.

5.4 Scheduler

The scheduler determines which ITasks and how many instances of them to run. If serialization done by the partition manager cannot alleviate the memory pressure, the scheduler will reduce the number of task instances (Lines 15–17). The selection of ITask instances to interrupt is based on the following three rules:

- *MITask First Rule:* Threads running MITasks have the highest priority to continue running. Since an MITask often performs data merging, terminating the thread would create a large number of input/output fragments.
- *Finish Line Rule*: A thread running an ITask closer to the finish line has a higher priority to continue to run.
- *Speed Rule*: For a set of threads running the same ITask, the slowest thread will be terminated first. The processing speed of a thread is determined by the number of `scaleLoop` iterations executed between two consecutive memory usage checks (performed by the monitor).

When a thread is selected to be interrupted, for this thread, the `ITaskScheduler.terminate(this)` method call (Line 24 in Figure 4) will return true and its `interrupt` method will be executed. The scheduler continues to terminate threads until the memory usage goes below the threshold. Upon receiving a "GROW" signal from the monitor, the scheduler creates a new thread to run an ITask based on the following two rules (Lines 20–23):

- *Spatial Locality Rule:* We favor an ITask that has *in-memory* inputs. These partitions can be processed first before the manager needs to load partitions from disk.
- *Finish Line Rule:* We favor an ITask that is closer to the finish line. When an ITask is selected and its input partition is on disk, the partition manager loads the partition back into memory transparently to the scheduler.

6. Evaluation

We have implemented the ITask library and the IRS on Hadoop and Hyracks. These implementations have approximately 30,000 lines of Java code. We ran Hadoop and Hyracks on a 11-node Amazon EC2 cluster. Each node (a c3.2x large instance) has 2 quad-core Intel Xeon E5-2680 2.80GHz processors, 15GB of RAM, and one RAID-0 comprised of 2 80GB SSDs. The cluster runs Linux 3.10.35 with enhanced networking performance. We used Java HotSpot(TM) 64-bit Server VM (build 24.71-b01) for all experiments. The state-of-the-art parallel generational garbage collector was used.

Name	MSA	IMC	IIB	WCM	CRP
Data	StackOverflow	Wikipedia	Wikipedia	Wikipedia	Wikipedia SP
Size	29GB	49GB	49GB	49GB	5GB
MH, RH	1GB, 1GB	0.5GB, 1GB	0.5GB, 1GB	0.5GB, 1GB	1GB, 1GB
MM, MR	6, 6	13, 6	13, 6	13, 6	6, 6
CTime	1047	5200	1322	2643	567
PTime	**48**	337	2568	2151	6761
ITime	72	**238**	**1210**	**1287**	**2001**

Table 1. Hadoop performance comparisons for five real-world problems we have reproduced: Map-Side Aggregation (MSA) [13], In-Map Combiner (IMC) [16], Inverted-Index Building (IIB) [8], Word Cooccurrence Matrix (WCM) [15], and Customer Review Processing (CRP) [10]. Reported are the name of each program (*Name*); the dataset used (*Data*) and its size (*Size*); the developer-reported Hadoop configuration including the max heap size for each Map and Reduce task (*MH* and *RH*), the max #Mappers and Reducers (*MM* and *MR*); the time elapsed before OME occurs in the original program (*CTime* in seconds); the time taken for the program to finish when the fix recommended on StackOverflow was used (*PTime* in seconds); and finally the running time for its ITask version (*ITime* in seconds). Highlighted are the lowest running times for the successful executions. SP represents a sample of the full dump; the StackOverflow data dump has a total of 25.8M posts, the Wikipedia data dump has a total of 4.7M articles; Wikipedia SP is a sample of Wikipedia data dump with 490K articles. In these experiments, the HDFS block size is 128MB.

Methodology Since Hadoop is a popular data-parallel framework that has many OMEs reported on StackOverflow, we focus our first set of experiments (§6.1) on reproducing real-world problems in Hadoop and understanding whether the ITask implementations can help these programs survive OMEs and successfully process their entire datasets. The second set of experiments (§6.2) focuses on comparing performance and scalability between the ITask programs and their original versions on Hyracks over various heap configurations and data sizes. For both Hadoop and Hyracks, we used their latest versions (2.6.0 and 0.2.14) in our experiments. YARN was enabled when Hadoop was run.

6.1 ITasks in Hadoop

We have successfully reproduced and implemented ITasks for 13 problems among the 73 problems we have studied. On average, it took us about a week to set up a distributed configuration as described on StackOverflow, manifest the problem, understand its semantics, and develop its ITask version. For all of these 13 problems [5–17], their ITask versions successfully survived memory pressure and processed the given datasets. Due to space limitations, we report detailed experimental results for a diverse array of 5 problems.

Table 1 shows these 5 problems and the configurations in which they manifest. Time elapsed before each program ran out of memory is also reported (in Row *CTime*). For each problem, we carefully read its recommended fixes on StackOverflow. For all the problems but CRP, the recommended fixes were changing parameters (# Map/Reduce workers on each node or task granularity). After a long and tedious tuning process, we observed that reducing worker numbers and/or sizes of data splits was indeed helpful. For these four problems, we ended up finding configurations under which the programs could successfully run to the end. The *shortest* running time under different working configurations we found is reported in Row *PTime*. For CRP, since the recommended fix was to break long sentences, we developed a tool that automatically breaks sentences whose length exceeds a threshold.

Name	Processed Input	Final Results	Intermediate Results	Lazy Serialization
MSA	14.9K	33.7G	0	6.0G
IMC	18.4K	23.1G	0	0
IIB	70.1M	0	7.1G	2.3G
WCM	192.6M	0	14.3G	1.5G
CRP	1.0K	1.2G	112.8M	0

Table 2. A detailed breakdown of memory savings from releasing different parts of the memory consumption.

Since we are not experts in natural language processing, this tool broke sentences in a naïve way and might be improved when considering domain knowledge.

The ITask versions of these problems were executed under the same Hadoop configuration as their original versions (as shown in Table 1). Their running time is shown in Section *ITime*. Comparing *PTime* and *ITime*, we can observe that the ITask versions have much better performance (i.e., on average 2× faster) than manually-tuned versions in most cases. The only exception is for MSA, where its *ITime* is 1.5× longer than *PTime*. An investigation identified the reason: since the first Map task loads a very large table to perform hash join, the program has to be executed with only a small degree of parallelism. Manual tuning sets the maximum number of workers to 1 thus paying no additional runtime cost while its ITask version alternates the number of workers between 1 and 2—the tracking overhead cannot be offset by the exploited parallelism. Another observation is that *CTime* may be much longer than *PTime* and *ITime*. This is because these programs all suffered from significant GC overhead as well as many restarts before YARN eventually gave up retrying and reported the crash.

Memory savings breakdown Table 2 shows a detailed breakdown of memory savings from releasing various parts of an ITask's consumed memory. Note that different programs have different semantics and therefore they benefit from different optimizations. For instance, the OME in MSA occurs in a Map task; the task has an extremely large key-value buffer, which contains final results that can be pushed to the next stage. Hence, MSA benefits mostly from pushing out and releasing final results. As another example, WCM crashes in a Reduce task; therefore, it has large amounts of intermediate results that can be swapped out and merged later. These promising results clearly suggest that ITask is effective in reducing memory pressure for programs with different semantics and processing different datasets.

With these programs, we have also compared ITask executions with naïve techniques that (1) kill a task instance upon memory pressure and later reprocess the same data partition from scratch (without using ITasks) and (2) randomly pick threads to terminate and data partitions to resume (without using our priority rules in §5.4). The results show that the ITask executions are up to 5× faster than these naïve techniques. Details of the comparison are omitted.

6.2 ITasks in Hyracks

The goal of this set of experiments is to understand the improvement in performance and scalability ITask provides for a regular data-parallel program. The 11-node cluster was still used; unless specified, each node used a 12GB heap as the default configuration.

Benchmarks We selected the following five already hand-optimized applications from Hyracks' code repository and ported their tasks to ITasks. These programs include word

Size	#Vertices	#Edges
72GB	1,413,511,390	8,050,112,169
44GB	992,128,706	4,474,491,119
27GB	587,703,486	2,441,014,870
14GB	143,060,913	1,470,129,872
10GB	75,605,388	1,082,093,483
3GB	24,973,544	313,833,543

Table 3. The inputs for WC, HS, and II: the Yahoo! Webmap (72GB) and its subgraphs.

Scale	Size	#Customer	#Order	#LineItem
150×	150.4GB	2.25×10^7	2.25×10^8	9.00×10^8
100×	99.8GB	1.50×10^7	1.50×10^8	6.00×10^8
50×	49.6GB	7.50×10^6	7.50×10^7	3.00×10^8
30×	29.7GB	4.50×10^6	4.50×10^7	1.80×10^8
20×	19.7GB	3.00×10^6	3.00×10^7	1.20×10^8
10×	9.8GB	1.50×10^6	1.50×10^6	6.00×10^7

Table 4. The inputs for HJ and GR: TPC-H data.

Name	DS	#K	#T
Word Count (WC)	14GB	2	32KB
Heap Sort (HS)	27GB	6	32KB
Inverted Index (II)	3GB	8	16KB
Hash Join (HJ)	100×	8	32KB
Group-By (GR)	50×	6	16KB

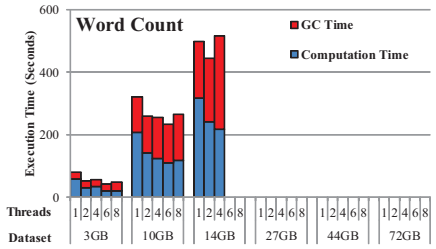
Table 5. The scalability of the original programs under a 12GB Java heap: *DS* reports the largest datasets in our experiments to which the programs scaled; *#K* and *#T* report the numbers of threads and the task granularities for which the best performance was obtained when processing the datasets shown under *DS*.

count (WC), heap sort (HS), inverted index (II), hash join (HJ), and group-by (GR). Note that these applications were selected because (1) they provide a basis for many high-level applications built on top of Hyracks, such as AsterixDB [1, 20] and Preglix [27]; and (2) they were used extensively in prior work [22, 24, 25] to evaluate Hyracks and other high-level applications. Since Hyracks does not allow the use of Java objects, these programs are already well-tuned and expected to have high performance.

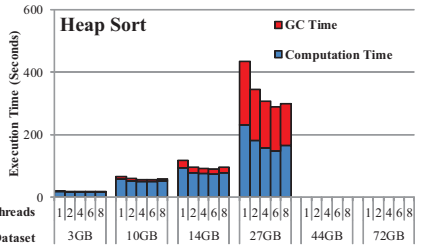
On average, each program has around 2K lines of Java code—e.g., the smallest program WC has 550 LOC and the largest program HJ has 3.2K LOC. It took us one person week to convert these five programs into ITask programs.

Our datasets came from two sources: the Yahoo Webmap [55], which is the largest publicly available graph with 1.4B vertices and 8.0B edges, and the TPC-H data generator [52], which is the standard data warehousing benchmark tool popularly used in the data management community. For TPC-H, we used the following three tables: Customer, Order, and LineItem. Table 3 and Table 4 show their statistics.

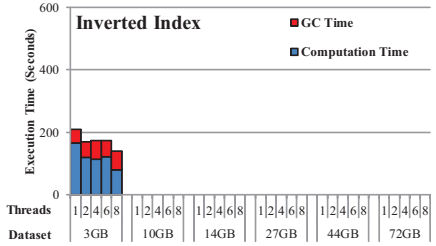
Scalability of the original programs We first ran each original version with various numbers of threads (between 1 and 8). Detailed performance comparisons among these configurations are shown in Figure 9. The configurations in which the program ran out of



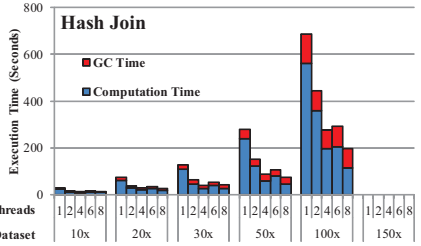
(a) WC failed on the 27GB, 44GB and 72GB datasets.



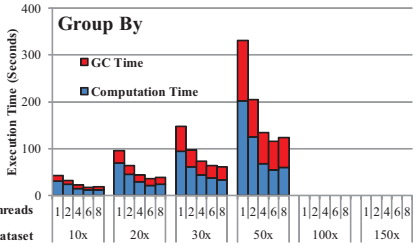
(b) HS failed on the 44GB and 72GB datasets.



(c) II failed on all the datasets except the 3GB one.



(d) HJ failed on the 150× dataset.



(e) GR failed on the 100× and 150× datasets.

Figure 9. Performance changes as we vary thread number in the original programs; the task granularity is 32KB.

memory are omitted. In each graph, bars are grouped under input sizes. From left to right, they show the execution times with growing numbers of threads. For each bar, the time is further broken down to the GC time (the upper part) and the computation time (the lower part). These graphs clearly demonstrate that increasing thread number does not always lead to better performance. For example, for HS and GR at their largest inputs (27GB and 50×, respectively), their fastest executions used 6 threads.

We have also varied task granularities (between 8KB and 128KB). Table 5 summarizes the largest datasets in our experiments to which these programs could scale and the configurations under which the best performance was obtained over these datasets. Most of these programs suffered from significant GC effort when large datasets were processed. For instance, for HS and GR, their GC time accounts for 49.14% and 52.27% of their execution time, respectively.

Among the programs, II has the worst scalability due to the large in-memory maps it maintains: II was only able to process the smallest dataset (3GB). Even the single-threaded

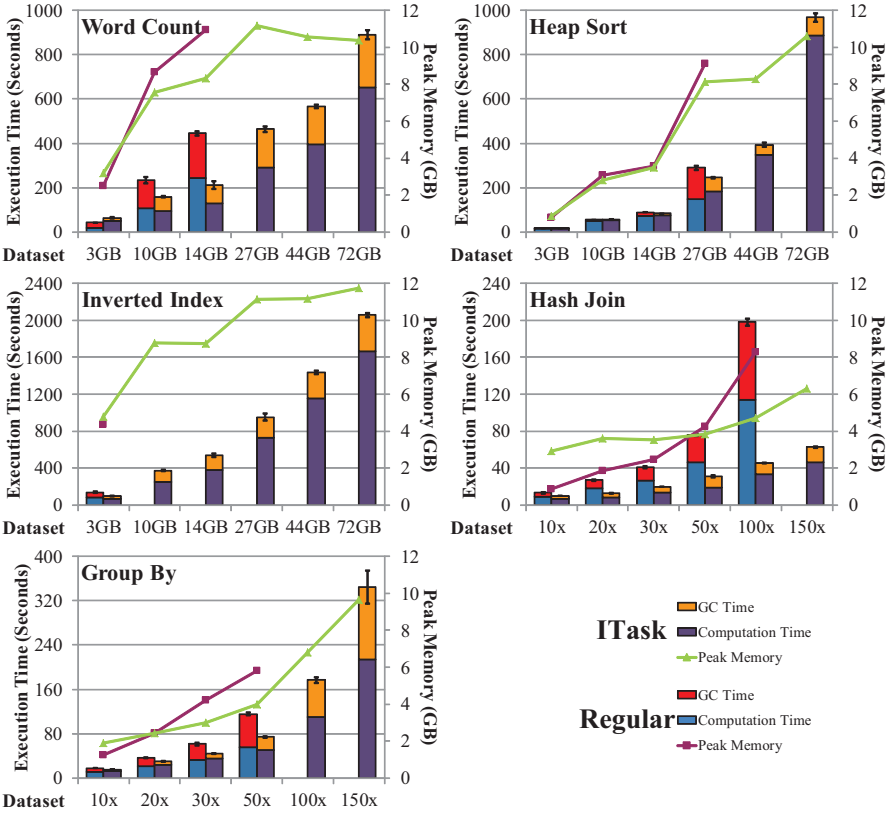


Figure 10. Comparisons between the ITask versions (the second/right bar in each pair) and their Java counterparts with the best configurations (the first/left bar). Each error bar represents the standard deviation of the times collected from 5 runs.

version of II could not process the 10GB dataset on the cluster. HJ scales the best: each slave was able to process up to 10GB input with a 12GB heap.

Performance improvements For each program, we next compare its ITask version with the original version under the configuration that yields the best performance (as shown in Table 5). We have measured both running time and heap consumption in this experiment. To eliminate the execution noise on the cluster, we ran the ITask version 5 times with a 12GB heap. Figure 10 reports the geometric means of these measurements. Bars represent running time of successful executions and are grouped by input sizes. In each group, the first/left bar corresponds to the best configuration for the original program and the second/right bar corresponds to the ITask version. Each bar is also broken down into GC (the upper part) and computation time (the lower part). The heap consumptions are represented by lines, each reporting the maximum heap usage of the program across all the slaves.

Table 6 summarizes the time and space savings from ITask. Among the 30 (both failed and successful) executions of these programs, their ITask versions were faster than their original versions in 27 of them. The 3 executions (for WC and HS) in which the ITask version was slower all processed very small datasets, plus the time differences are negligible (i.e.,

Name	#TS	%TS	#HS	%HS	Scalability
WC	5/6	39.63%	5/6	13.81%	5.14×
HS	4/6	10.85%	5/6	7.57%	2.67×
II	6/6	27.53%	5/6	-9.28%	24.00×
HJ	6/6	66.45%	3/6	-5.16%	6.00×
GR	6/6	61.35%	5/6	26.62%	5.00×
GeoMean	27/30	44.95%	23/30	7.65%	6.29×

Table 6. A summary of the performance improvements from ITask: *#TS* and *#HS* report ratios at which an ITask program outperforms its regular counterpart in execution time and heap consumption, respectively; *%TS* and *%HS* report the ITask's reductions in time and heap consumption, respectively, for the inputs both versions have successfully processed; *Scalability* reports the ratios between the sizes of the largest datasets the two versions can scale to.

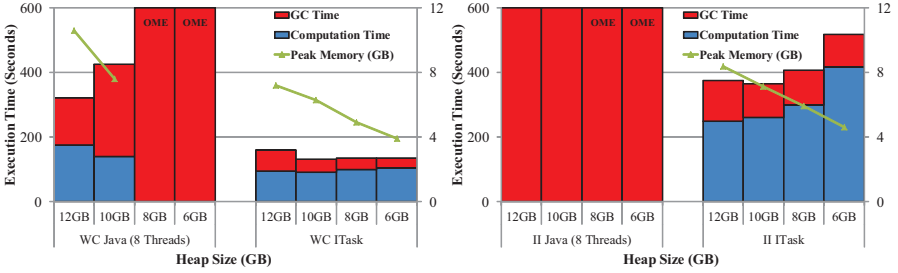
1.61%). The average time reduction ITask has achieved across the 17 successful executions is 44.95%. The majority of these savings stems from significantly reduced GC costs.

The ITask programs are also memory-efficient. In 23/30 executions, the maximum heap consumption is smaller than that of the Java version. This is due to IRS's ability to move data in and out. However, in the cases that the inputs are small, the ITask version consumes more memory because of the tracking/bookkeeping performed in the IRS. The overall memory space reduction across the executions in which both versions succeeded is 7.65%; furthermore, the original programs failed in 13 out of the 30 executions while the ITask version succeeded in all of them.

Scalability improvements The last column of Table 6 shows how well the ITask programs scale. These measurements are computed as the ratios between the sizes of the largest inputs the ITask-based and the original programs can process. As shown in Figure 10, all the ITask programs successfully processed all input sizes in our experiments while none of the original programs could. Even for the highly-scalable HJ program, its ITask version well outperforms its original version. Overall, ITask has achieved a 6.29× scalability improvement. We performed an additional test on further larger input sizes to understand the scalability upper bound of these programs. This experiment shows that the ITask versions of HJ and GR could successfully process a 600× and a 250× dataset, respectively. These results indicate that the scalability improvement ITask provides may be even larger when bigger datasets are used.

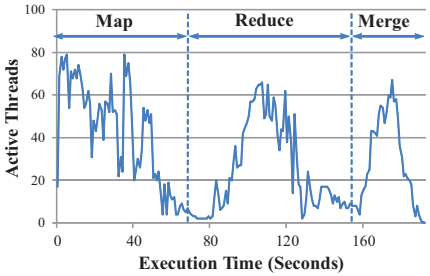
Using different heaps To understand how an ITask program behaves under different heaps, we ran WC and II on the 10GB dataset, under a 12GB, 10GB, 8GB, and 6GB heap. Their detailed performance is shown in Figure 11 (a) and (b). To summarize, when the input size is fixed, the performance of an ITask program does not change much with the heap size, while a Java program can easily crash when the heap size is reduced. For example, the original program of WC could not process the 10GB dataset with the 8GB and 6GB heap, while its ITask version successfully processed the whole dataset with the 6GB heap, yielding a running time comparable to that with the 10GB heap. In addition, the GC component is less than 10% of the total time.

In order to closely examine ITask's adaptive execution, we counted the number of active ITask instances during the execution of WC on the 14GB dataset. Figure 11 (c) shows how the number of threads changes as the execution progresses on the cluster. The cluster has a maximum of 80 workers. Threads for Map and Reduce can overlap. The program



(a) WC on the 10GB dataset

(b) II on the 10GB dataset



(c) WC on the 14GB dataset

Figure 11. (a) and (b) show how performance changes as we vary the heap size; (c) shows how the number of active ITask instances changes as the execution progresses.

finished in 192 seconds and the average number of active threads on each slave was 3.16. Figure 11 (c) clearly shows that an ITask execution is very dynamic and our runtime system can automatically adapt the active worker numbers to memory availability while keeping as many active workers as possible. It would be interesting to also measure the cost of disk I/O. However, we create background threads for disk operations, making it difficult to separate out the I/O cost. Writing data partitions occurs simultaneously with the data processing while reading data can introduce stalls. These stalls contribute to 5-8% of the execution time.

7. Related Work

Data-parallel systems MapReduce [33] has inspired a body of research on distributed data-parallel computation, including Hyracks [4], Hadoop [21], Spark [60], or Dryad [38]. The MapReduce model has been extended [56] with Merge to support joins and adapted to support pipelining [32]. Yu *et al*. propose a programming model [57] for distributed aggregation for data-parallel systems.

A number of high-level declarative languages for data-parallel computation have been proposed, including Sawzall [49], Pig Latin [48], SCOPE [28], Hive [51], and DryadLINQ [58]. All of these frameworks and languages except SCOPE and Dryad were implemented in JVM-based languages such as Java and Scala and thus can immediately benefit from the ITask optimization proposed in this paper. SCOPE and Dryad were implemented in C#, which also runs on top of a managed runtime system; we expect the ITask idea can also be adapted to optimize their applications.

Optimizations of data-parallel systems While there exists a large body of work on optimizing data-parallel systems, most existing efforts focus on domain-specific optimizations, including, for example, data pipeline optimizations [29, 35, 62], query optimizations [32, 46], or shuffling optimizations [42, 51, 61]. Despite these optimizations, Big Data performance is still fundamentally limited by memory inefficiencies inherent in the underlying programming systems. ITask is the *first attempt* to help data-parallel tasks written in a managed language survive memory pressure and scale to large datasets by providing a programming model for developers to reason about interrupts and a runtime system that interrupts tasks and tunes performance.

Cascading [2] is a Java library built on top of Hadoop. It provides abstractions for developers to explicitly construct a dataflow graph to ease the challenge of programming data-parallel tasks. Similarly to Cascading, FlumeJava [29] is another Java library that provides a set of immutable parallel collections. These collections present a uniform abstraction over different data representations and execution strategies for MapReduce. StarFish [36] is a self-tuning framework for Hadoop that provides multiple levels of tuning support. At the heart of the framework is a Just-In-Time optimizer that profiles Hadoop jobs and adaptively adjusts various framework parameters and resource allocation. ITasks perform autotuning in orthogonal way: it is not bound to a specific framework nor is limited to a specific task semantics. Instead, it provides a generic way to reduce memory pressure for a variety of different frameworks and tasks.

Resilient Distributed Datasets (RDD) [59] provides a fault tolerant abstraction for managing datasets in a distributed environment. It is similar to ITask in that the physical location of a data structure is transparent to the developer. However, ITask scatters data between memory and disk on each machine while RDD distributes data in the cluster. Moreover, ITask focuses on enabling managed tasks to survive the presence of high memory pressure while RDD focuses on data recovery in the presence of node failures.

Spark [60] implements RDD and divides jobs into "stages". While resource contention can be avoided between stages, memory problems can still occur inside each stage. In Spark, RDDs can be spilled to disk, but the spilling mechanism is much less flexible than ITask: when spilling is triggered, all RDDs with the same key need to be spilled; partial spilling is not possible.

Mesos [37] and YARN [3] provide sophisticated resource management that can intelligently allocate resources among different compute nodes. Although these job schedulers have a global view of the resources on the cluster, their resource allocation is semantics-agnostic and based primarily on resource monitoring. However, the memory behavior of a program on each node is very complex and can be affected by many different factors. Hence, memory pressure still occurs, impacting application performance and scalability. ITask is designed to bring the execution back to the safe zone of memory usage when pressure arrives.

PeriSCOPE [35] is a system that automatically optimizes programs running on the SCOPE data-parallel system. It applies compiler-like optimizations on the declarative encoding of a program's pipeline topology. FACADE [47] and Broom [34] optimize the managed runtime by allocating data items in regions. While ITask aims to solve a similar memory problem, it does so by allowing tasks to be interrupted and using a runtime system to automatically interrupt/resume tasks, rather than eliminating Java objects.

8. Conclusions

We present interruptible tasks as a systematic approach to help data-parallel tasks survive memory pressure. ITask contains a novel programming model that can be used by developers to reason about interrupts as well as a runtime system that automatically performs interrupts and adaptation. Using real-world examples and experimental data, we demonstrate that (1) ITasks can be easily integrated into a distributed framework and interact seamlessly with the rest of the framework; and (2) the runtime is effective at reducing memory usage, thereby significantly improving the performance and scalability of a variety of data-parallel systems.

Acknowledgments We would like to thank our shepherd Luis Ceze as well as the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CCF-0846195, CCF-1217854, CNS-1228995, CCF-1319786, CNS-1321179, CCF-1409829, CCF-1439091, CCF-1514189, CNS-1514256, by the Office of Naval Research under grant N00014-14-1-0549, and by an Alfred P. Sloan Research Fellowship.

References

[1] AsterixDB. https://asterixdb.ics.uci.edu/.

[2] Cascading. http://www.cascading.org.

[3] Hadoop YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/.

[4] Hyracks. http://hyracks.org/.

[5] Out of memory error due to appending values to stringbuilder. http://stackoverflow.com/questions/12831076/.

[6] Out of memory error due to large spill buffer. http://stackoverflow.com/questions/8464048/.

[7] Out of memory error in a web parser. http://stackoverflow.com/questions/17707883/.

[8] Out of memory error in building inverted index. http://stackoverflow.com/questions/17980491/.

[9] Out of memory error in computing frequencies of attribute values. http://stackoverflow.com/questions/23042829/.

[10] Out of memory error in customer review processing. http://stackoverflow.com/questions/20247185/.

[11] Out of memory error in efficient sharded positional indexer. http://www.cs.cmu.edu/~lezhao/TA/2010/HW2/.

[12] Out of memory error in hash join using distributedcache. http://stackoverflow.com/questions/15316539/.

[13] Out of memory error in map-side aggregation. http://stackoverflow.com/questions/16684712/.

[14] Out of memory error in processing a text file as a record. http://stackoverflow.com/questions/12466527/.

[15] Out of memory error in word cooccurrence matrix stripes builder. http://stackoverflow.com/questions/12831076/.

[16] The performance comparison between in-mapper combiner and regular combiner. http://stackoverflow.com/questions/10925840/.

[17] Reducer hange at the merge step. http://stackoverflow.com/questions/15541900/.

[18] Tuning Spark. http://spark.apache.org/docs/latest/tuning.html.

[19] AHMAD, F., CHAKRADHAR, S. T., RAGHUNATHAN, A., AND VIJAYKUMAR, T. N. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *USENIX ATC* (2014), pp. 1–12.

[20] ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BORKAR, V. R., BU, Y., CAREY, M. J., CETINDIL, I., CHEELANGI, M., FARAAZ, K., GABRIELOVA, E., GROVER, R., HEILBRON, Z., KIM, Y., LI, C., LI, G., OK, J. M., ONOSE, N., PIRZADEH, P., TSOTRAS, V. J., VERNICA, R., WEN, J., AND WESTMANN, T. Asterixdb: A scalable, open source BDMS. *PVLDB 7*, 14 (2014), 1905–1916.

[21] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[22] BEHM, A., BORKAR, V. R., CAREY, M. J., GROVER, R., LI, C., ONOSE, N., VERNICA, R., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND TSOTRAS, V. J. ASTERIX: Towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases 29* (2011), 185–216.

[23] BOND, M. D., AND MCKINLEY, K. S. Leak pruning. In *ASPLOS* (2009), pp. 277–288.

[24] BORKAR, V. R., CAREY, M. J., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE* (2011), pp. 1151–1162.

[25] BORKAR, V. R., CAREY, M. J., AND LI, C. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT* (2012), pp. 3–14.

[26] BU, Y., BORKAR, V., XU, G., AND CAREY, M. J. A bloat-aware design for big data applications. In *ISMM* (2013), pp. 119–130.

[27] BU, Y., BORKAR, V. R., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB 8*, 2 (2014), 161–172.

[28] CHAIKEN, R., JENKINS, B., LARSON, P., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB 1*, 2 (2008), 1265–1276.

[29] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI* (2010), pp. 363–375.

[30] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[31] CHU, C. T., KIM, S. K., LIN, Y. A., YU, Y., BRADSKI, G. R., NG, A. Y., AND OLUKOTUN, K. Map-reduce for machine learning on multicore. In *NIPS* (2006), pp. 281–288.

[32] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce online. In *NSDI* (2010), pp. 313–328.

[33] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[34] GOG, I., GICEVA, J., SCHWARZKOPF, M., VASWANI, K., VYTINIOTIS, D., RAMALINGAM, G., COSTA, M., MURRAY, D. G., HAND, S., AND ISARD, M. Broom: Sweeping out garbage collection from big data systems. In *HotOS* (2015).

[35] GUO, Z., FAN, X., CHEN, R., ZHANG, J., ZHOU, H., MCDIRMID, S., LIU, C., LIN, W., ZHOU, J., AND ZHOU, L. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI* (2012), pp. 121–133.

[36] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *CIDR* (2011), pp. 261–272.

[37] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), pp. 295–308.

[38] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), pp. 59–72.

[39] Kryo. https://github.com/EsotericSoftware/kryo.

[40] KWON, Y., REN, K., BALAZINSKA, M., AND HOWE, B. Managing skew in hadoop. *IEEE Data Eng. Bull. 36*, 1 (2013), 24–33.

[41] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI* (2012), pp. 31–46.

[42] LIU, J., RAVI, N., CHAKRADHAR, S., AND KANDEMIR, M. Panacea: Towards holistic optimization of MapReduce applications. In *CGO* (2012), pp. 33–43.

[43] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB 5*, 8 (2012), 716–727.

[44] MITCHELL, N., SCHONBERG, E., AND SEVITSKY, G. Four trends leading to java runtime bloat. *IEEE Software 27*, 1 (2010), 56–63.

[45] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA* (2007), pp. 245–260.

[46] MURRAY, D. G., ISARD, M., AND YU, Y. Steno: Automatic optimization of declarative queries. In *PLDI* (2011), pp. 121–131.

[47] NGUYEN, K., WANG, K., BU, Y., FANG, L., HU, J., AND XU, G. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS* (2015), pp. 675–690.

[48] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In *SIGMOD* (2008), pp. 1099–1110.

[49] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[50] TANG, Y., GAO, Q., AND QIN, F. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX ATC* (2008), pp. 307–320.

[51] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive - A warehousing solution over a map-reduce framework. *PVLDB 2*, 2 (2009), 1626–1629.

[52] The TPC Benchmark(TM)H (TPC-H). http://www.tpc.org/tpch.

[53] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER* (2010), pp. 421–426.

[54] XU, G. H., MITCHELL, N., ARNOLD, M., ROUNTEV, A., SCHONBERG, E., AND SEVITSKY, G. Scalable runtime bloat detection using abstract dynamic slicing. *TOSEM 23*, 3 (2014), 23.

[55] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[56] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: Simplified relational data processing on large clusters. In *SIGMOD* (2007), pp. 1029–1040.

[57] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP* (2009), pp. 247–260.

[58] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), pp. 1–14.

[59] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012), pp. 15–28.

[60] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[61] ZHANG, J., ZHOU, H., CHEN, R., FAN, X., GUO, Z., LIN, H., LI, J. Y., LIN, W., ZHOU, J., AND ZHOU, L. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI* (2012), pp. 22–22.

[62] ZHOU, J., LARSON, P.-Å., AND CHAIKEN, R. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE* (2010), pp. 1060–1071.