Transformer: A New Paradigm for Building Data-Parallel Programming Models

Cloud computing drives the design and development of diverse programming models for massive data processing. The Transformer programming framework aims to facilitate the building of diverse data-parallel programming models. Transformer has two layers: a common runtime system and a model-specific system. Using Transformer, the authors show how to implement three programming models: Dryad-like data flow, MapReduce, and All-Pairs.

Peng Wang
Dan Meng
Jizhong Han
Jianfeng Zhan
Bibo Tu
Institute of Computing Technology, Chinese Academy of Sciences
Xiaofeng Shi
Le Wan
Tencent Corporation

Recently, driven by a huge increase in dataset size, we've witnessed the development of various scalable software infrastructures for massive data processing. Several data-parallel programming models have been proposed to solve domain-specific applications. For example, MapReduce is best at handling group-by-aggregation applications.[1] Hadoop (http://hadoop.apache.org) provides a widely adopted open source Java implementation of MapReduce. Dryad is more general and flexible than MapReduce, as it allows the execution of arbitrary computation that can be expressed as a directed acyclic graph (DAG).[2] However, Dryad is not appropriate for applications such as iterative jobs, nested parallelism, and irregular parallelism.[3] All-Pairs, which is used in biometrics, bioinformatics, and data mining applications, aims to solve the problems that often arise when comparing element pairs in two data sets by applying a given function.[4] Google's Pregel specializes in large-scale graph computing, such as Web graph and social network analysis.[5] Furthermore, several high-level language systems, such as Google's Sawzall,[6] Yahoo's Pig Latin,[7] Facebook's Hive,[8] Microsoft's Scope,[9] and DryadLinq,[10] improve programmers' productivity. Future work might identify new patterns from emerging classes of applications and create other programming abstractions for processing large volume of data. Additionally, Patterson claims that there would be more datacenter programming languages if we consider MapReduce as the first instruction of the "datacenter computer."[11,12]

Building a programming model from scratch is demanding. First, distributed system development usually requires sophisticated expertise and thus has a high entry barrier. For example, distributed systems usually involve concurrency; however,

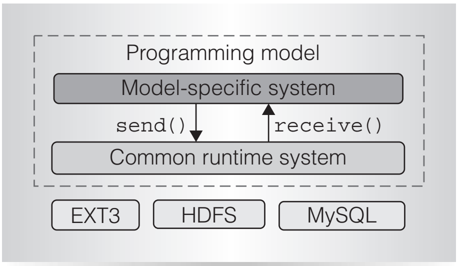

Figure 1. Transformer's layered architecture. Transformer uses various storage systems in the storage layer—from local file systems to distributed file systems (such as the Hadoop Distributed File System [HDFS]) and databases (such as MySQL). (Ext3: 3rd extended file system)

multithreaded concurrent programming is difficult and it's easy for programmers to make errors. Second, to ease the burden on application developers, a robust and scalable programming model must automatically handle various issues, such as task scheduling, fault tolerance, data distribution, and load balance. Achieving these goals in a system is complicated and requires a huge amount of effort and time.

Existing programming models are developed using C++ or Java libraries such as remote procedure call, serialization, and multithread. Although these libraries simplify the development of distributed systems to some extent, programmers must still master various APIs and deal with messy details (for example, exception handling). Furthermore, a programming model is usually designed for a specific programming abstraction and has a monolithic architecture. Therefore, from an engineering standpoint, adapting an existing programming model to a new one is time-consuming or even impossible.

To address these issues, we propose a paradigm for building programming models based on two concise primitives: `send()` and `receive()`. This paradigm led to the design and implementation of *Transformer*, a programming framework that lets us build diverse data-parallel programming models on a cluster of shared-nothing commodity machines in a uniform manner. Using Transformer, we've built three programming models—a Dryad-like data flow,[13] MapReduce, and All-Pairs to demonstrate the proposed paradigm. As our case studies show, our programming paradigm significantly reduces the complexity of building a programming model.

Design and implementation

Our target environment is a typical data center that deploys a cluster of hundreds or thousands of commodity servers to process large-scale data. Presently, we collaborate with Tencent (http://www.tencent.com), which provides the largest Internet-based instant message service (QQ) in the world, with more than one billion registered users.

Design principle

In general, a programming model implements a specific computational model that can be leveraged to solve targeted applications. Here, *computational model* refers to a programming abstraction derived from distributed applications. MapReduce's key/value pairs and Dryad's DAG are typical examples. Commonalities exist among data-parallel programming models. Most (if not all) programming models have two recurring patterns: running multiple tasks on different machines and delivering data sets to where they are needed. Thus, it's necessary to factor these elements into a common layer. We adopt the "separation of mechanism and policy" design principle to divide a programming model into two loosely coupled layers: the *common runtime* system (hereafter referred to as the *runtime*) and the *model-specific* system, shown in Figure 1.

The runtime is a substrate system that implements two commonly occurring data-intensive computing operations: executing tasks on machines and transporting data between machines. The runtime is provided as a library, from which programmers implement the model-specific system using two primitives (`send()` and `receive()`). The model-specific system's implementation depends on a particular computational model. It typically handles model-specific issues such as data partition, mapping tasks onto physical resources, data dependencies, and load balancing.

Table 1. Primitives in the Transformer programming framework.

API	Parameter	Return value	Mode	Command messages and responses: Task execution	Command messages and responses: Data transport
`send()`	Message	True/False	Nonblocking	`RunTask`	`MoveData`
				`KillTask`	`DelData`
`receive()`	None	Response	Blocking	`RunDone/RunFailed`	`MoveDone/MoveFailed`
				`KillDone/KillFailed`	`DelDone/DelFailed`

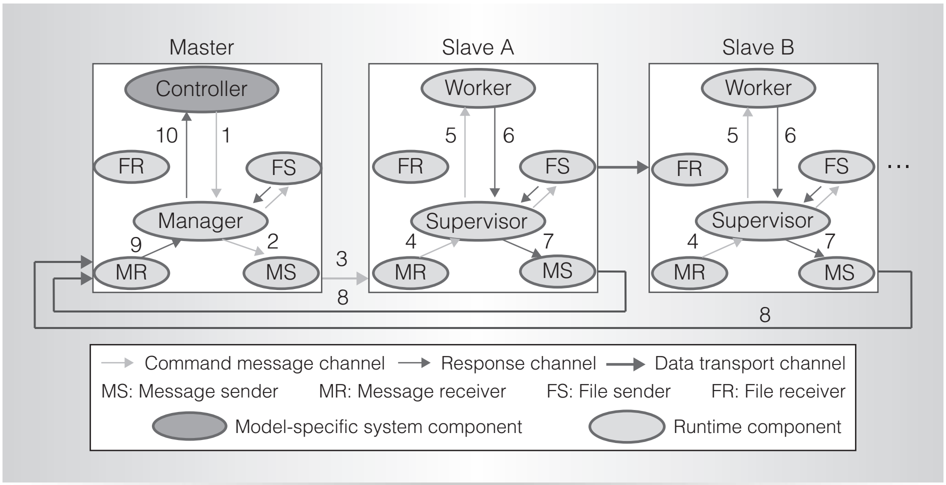

Figure 2. Transformer architecture. The Transformer programming framework consists of loosely coupled components.

Basic primitives

In our design, `send()` and `receive()` are powerful primitives for task execution and data transport. Programmers use `send()` to issue a command message describing an operation, and `receive()` to get a response indicating an operation's status: completed or failed. There are two classes of command messages: *task* and *data*. Table 1 describes the two primitives, the command messages, and their corresponding responses.

System architecture

As Figure 2 shows, Transformer adopts a master-slave structure. The runtime includes several major components: *manager*, *supervisor, worker*, *file sender/receiver,* and *message sender/receiver*. The model-specific system only contains one component: *controller*.

The Transformer system has two types of nodes: a *master* node (a centralized management node) and *slave* nodes (compute nodes). On a master node, the manager dispatches messages among different components and tracks each slave node's status (living or dead). Meanwhile, on each slave node, a supervisor dispatches messages to destined components and sends heartbeats to the manager indicating the slave node's status.

Every node has two communication components—message sender and message receiver—that perform internode communication. The message sender encodes a message into a byte stream and sends the byte stream to a specified destination. The message receiver receives a byte stream over the network, decodes the received data into a message, and forwards the received message to its related component (manager or supervisor). Each node also has file sender and file receiver components, which transfer data among nodes. On receiving a command message (for instance, a message of type

MoveData), the file sender begins transferring specified files to destinations. The file receiver receives file streams coming from different nodes and saves them to specified locations. The file sender creates either a response of type MoveDone for a completed transportation or a response of type MoveFailed for a failed transportation, and sends the response to its related component (manager or supervisor).

When the worker receives a command message of type RunTask, it spawns the specified executable in a separate process, and then monitors the status of the forked process. The worker creates a response of type RunDone for a completed task or type RunFailed for a failed task, and sends the response to the supervisor.

Figure 2 summarizes how a command message and its response move through Transformer. Suppose we want to run a task on a machine (for example, Slave A). The controller first creates a task message of type RunTask and sends it to the manager via send(). The manager receives the command message and forwards it to the message sender. The latter sends this message to the message receiver on Slave A (the destined node for this message). The message receiver decodes the received data into a message and forwards the message to the supervisor, which dispatches the message to the worker in charge of launching the task. That is to say, the command message follows the path 1-2-3-4-5 in Figure 2. When the task completes successfully, the worker on Slave A create a response and sends it to the supervisor. Here, the response follows another path: 6-7-8-9-10 in Figure 2. Finally, the response is queued in the controller's internal first-in, first-out (FIFO) buffer. The controller could use receive() to fetch responses from this buffer. Data transport is handled in an analogous fashion.

Fault tolerance

Failures are unavoidable in distributed applications. In the context of data-intensive computation on a cluster of commodity computers, failures are norms rather than exceptions. A programming model must tolerate failures gracefully without requiring application developers to handle them.

Transformer has an agile fault tolerance strategy. The runtime is only responsible for failure detection while the model-specific system is responsible for failure recovery, which usually requires re-executing a task or retransmitting data policies. In the collaboration of the runtime and the model-specific system, the former ensures that the latter is informed when task execution or data transport fails. At present, the runtime detects two types of failure—task failure and machine failure—which are monitored by two distinct components: worker and manager. For example, if a running task crashes (for example, a task on Slave A in Figure 2 crashes due to a read error on a bad record), the worker will create a Run-Failed response, which follows the path 6-7-8-9-10 in Figure 2 and finally reaches the controller's internal FIFO buffer. Once the controller receives this response, it can choose another available machine and re-execute the failed task by sending another RunTask command message. If a machine crashes for any reason, the manager will not receive heartbeats from the supervisor on that machine, so will mark this machine as unavailable and notify the controller, which will begin fault recovery operations.

Our fault tolerance design decision has two benefits. First, it decreases the design and implementation complexity in the runtime, thus increasing the robustness and reliability of the system core. Second, it provides the flexibility of model-specific systems, since developers can customize diverse fault-recovery policies according to their needs.

System implementation

We implemented Transformer, including its associated programming models, in approximately 5,000 lines of Python code. We employ the actor model,[14] instead of traditional multithreaded programming, to handle concurrency. As an effective approach to concurrency, the actor model has proven remarkably successful in languages such as Erlang (http://www.erlang.org) and Scala (http://scala-lang.org). It uses message passing to exchange states, as opposed to threaded approaches such as conditions and semaphores. An actor communicates with others through asynchronous message passing. Messages are delivered to an actor's mailbox. An actor is not blocked when it sends a message but is blocked if it calls the `receive()` method. Message passing concurrency is potentially more secure than shared memory concurrency with locks, as accessing an actor's mailbox is race-free by design.

We implemented the actor model in a library, which considerably simplifies the development of the overall system and ensures its correct implementation. In the Transformer library, we defined a Python base class with built-in `send()` and `receive()` methods for intercomponent communication. All Transformer components, except file receiver (which is a separate daemon) inherit from the base class. Components are loosely coupled through message passing instead of tightly coupled, allowing further optimization of individual components.

Because communication between the master and slave nodes is frequent, we adopted asynchronous network programming for the message receiver to achieve high concurrent network I/O. To reduce network overhead, the message sender further compresses a serialized message before sending it through a TCP socket.

Programming models

Because the controller is the only component that programmers need to customize, it's fairly easy to build a new programming model; developers just need to write model-specific code in the controller component.

We defined a Python base class in the Transformer library, from which a customized controller component inherits. The base class has three primary methods: `send()`, `receive()`, and `main()`. The first two methods are for interacting with the runtime. Programmers write model-specific code by overriding the `main()` method invocated by the system. The Transformer library also defines command message objects such as `RunTask` and `MoveData`. Attributes such as executable, invocation parameters, and file path can be set by calling methods defined in command message objects.

Our proposed approach differs from previous efforts in three ways.

First, with our approach, programmers need only focus on high-level issues when writing controller code, such as automatic data partition, scheduling multiple task instances on different machines, and fault recovery. Programmers do not need to deal with low-level issues: concurrency, remote procedure call, serialization, network programming, and so forth.

Second, Transformer allows further extension and enables the implementation of new features. For example, existing programming models such as MapReduce and Dryad adopt a backup task mechanism to alleviate the straggler problem (degenerately slow tasks).[1] This usually requires knowing the progress status of running tasks. Although the current implementation does not support this feature, the worker can communicate with the forked process using a named pipe in the file system. With the pipe, a task can report progress status to the worker, which creates and forwards a `TaskStatus` response to the supervisor. Thus, the controller could receive periodic status updates for tasks.

Third, from an engineering viewpoint, the controller is clean and short and allows further optimization with little effort. For example, in our three reference implementations, we've written 800 lines of Python code (controller codes) for data flow, 300 lines for MapReduce, and 100 lines for All-Pairs. Additionally, several programmers at Tencent have found it straightforward to further optimize our implementation and plan to build a production system with reference to our Transformer implementation.

Case study 1: Data-flow model

The first programming model we built is a Dryad-like data flow. According to data-flow principles, each application is modeled as a DAG of operators interconnected according to explicit declarations of data dependencies.

A data-flow job consists of one or more stages. Each stage corresponds to an element in the basic execution plan. The stage topology can be seen as a skeleton of the overall job. Every task inhabiting a vertex is placed in a stage of the data-flow graph. Tasks within a stage are independent—in other words, there is no data precedence constraint between them, and all tasks within a stage

```
# initialization
stageList = stageTopology(job.DAG)
msgDict = {}
# stage-by-stage scheduling
for stage in stageList:
   # send all messages of RunTask type in a stage
   for taskID in stage:
     runTaskMsg = genRunMsg(taskID)
     msgDict[taskID] = []
     msgDict[taskID].append(runTaskMsg)
     send(runTaskMsg)
   # wait until all tasks in a stage complete
   while not isStageFinished (stage):
     recvMsg = receive()
     taskID = parse(recvMsg)
     if recvMsg is RunDone:
         moveDataMsg = genMoveMsg(taskID)
         msgDict[taskID].append(moveDataMsg)
         send(moveDataMsg)
     if recvMsg is MoveDone:
         msgDict[taskID].append(recvMsg)
```

Figure 3. Implementation of data-flow model. A data-flow job consists of one or more task stages. Data flow is a stage-by-stage computation.

can run parallel if every task in the stage is assigned a computing node.

For clarity, we present a simplified implementation of the data-flow model in Figure 3. We assume that the controller mapped a logical execution plan onto physical resources prior to execution. In our controller implementation, we do not write code to deal with fault-recovery issues.

We apply a topological sorting on the job. The DAG objective is to obtain a stage list. The controller maintains a hash table (`msgDict`) that stores messages for a task. Each stage consists of two steps: scheduling tasks and transferring data. In a stage, the controller sends all `RunTask` command messages and fetches incoming responses by calling `receive()`. When it receives a `RunDone` response, indicating that a task has completed and output the desired data, the controller sends a `MoveData` command message to instruct the runtime to transfer data to specified locations.

Case Study 2: MapReduce model

MapReduce essentially provides an abstraction of group-by-aggregation operations over a cluster of machines. A programmer provides a map function that performs grouping and a reduce function that performs aggregation. MapReduce computation consists of four phases: map, shuffle, sort, and reduce. Map, sort, and reduce phases run multiple tasks in parallel, while the shuffle phase moves data between machines. In our reference implementation, we've supplied map and reduce Python classes to programmers. MapReduce computation is organized into three task stages: map, sort, and reduce. A sort task is followed by a reduce task running in the same node.

We use a simple data partition strategy for map tasks: input files are split into *M* partitions of approximately equal size and each map task processes one partition. Map tasks read data from `stdin` using Hadoop's command-line utility. A custom hash function partitions the outputs of map tasks into *R* regions. In the controller for MapReduce, we do not write code to deal with fault recovery or provide backup task mechanisms to deal with stragglers.

Case Study 3: All-Pairs Model

We've also built an All-Pairs model, which has a different nature from both data flow and MapReduce. All-Pairs abstraction aims to compute the Cartesian product of

```
srcList, destList = init()
roundCount = 0
while len(destList) > 0:
 roundCount +=1
 srcNums = len(srcList)
 destNums = len(destList)
 if srcNums <= destNums:
    for i in range(srcNums):
       src = srcList[i]
       dest = destList.pop()
       moveDataMsg = genMoveMsg(src,dest)
       send(moveDataMsg)
    counter = 0
    while counter != srcNums:
       recvMsg = receive()
       dest = parse(recvMsg)
       srcList.append(dest)
       counter += 1
 else:
    for i in range(destNums):
       src = srcList[i]
       dest = destList.pop()
       moveDataMsg = genMoveMsg(src,dest)
       send(moveDataMsg)
    counter = 0
    while counter != destNums:
       recvMsg = receive()
       dest = parse(recvMsg)
       srcList.append(dest)
       counter += 1
```

Figure 4. Implementation of a spanning tree algorithm. In each round, a source node is in charge of pushing data to a destination node without data.

two sets, generating a matrix in which each cell **M**[*i*][*j*] contains the output of the function F on object *A*[*i*] and *B*[*j*].

```
AllPairs( set A, set B, function F )
     Returns matrix M where M[i] [j]
       = F( A[i], B[j] ) for all i, j
```

The All-Pairs model targets a common pattern that appears throughout science and engineering. For example, in biometrics one might compute All-Pairs to determine the accuracy of a matching algorithm on a collection of face images. All-Pairs computation contains four steps: model the system, distribute the data, dispatch batch jobs, and clean up the system. The most difficult part of All-Pairs computation is determining how to quickly distribute a data set to many nodes. Moretti and colleagues propose a spanning tree algorithm to solve the distributing data problem.[4] We present how to implement the algorithm based on `send()` and `receive()` in Figure 4.

Evaluation

We performed all experiments on a 70-node cluster at Tencent. Each node had two dual-core Intel Xeon 3.00-GHz CPUs (in other words, four CPUs total) running the 32-bit SUSE 10 Linux (kernel version 2.6.16.60) with 4 Gbytes of DDR2 RAM and four SATA hard disks (280 Gbytes total). The nodes were connected to a Cisco switch via Broadcom NetXtreme BCM5704 Gbit Ethernet. We used Hadoop version 0.19.2 running on Java 1.6.0_16 and deployed Hadoop with the default configuration. We used version 2.4.2 of the Python interpreter.

In our experiments, the maximum run time of jobs was only 15 minutes and no failure happened on the scale of 70 nodes.

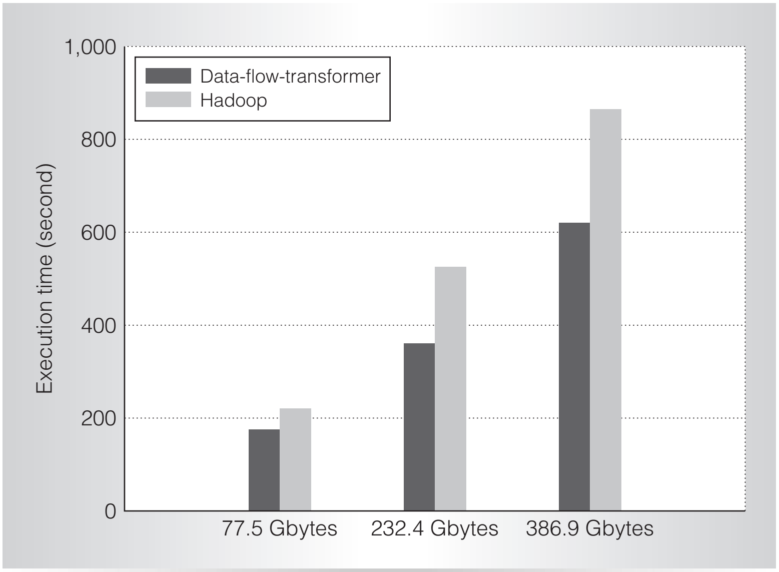

Figure 5. Experimental results from clickstream analysis. Columns represent the execution time of Hadoop and Data-flow-transformer. Lower is better.

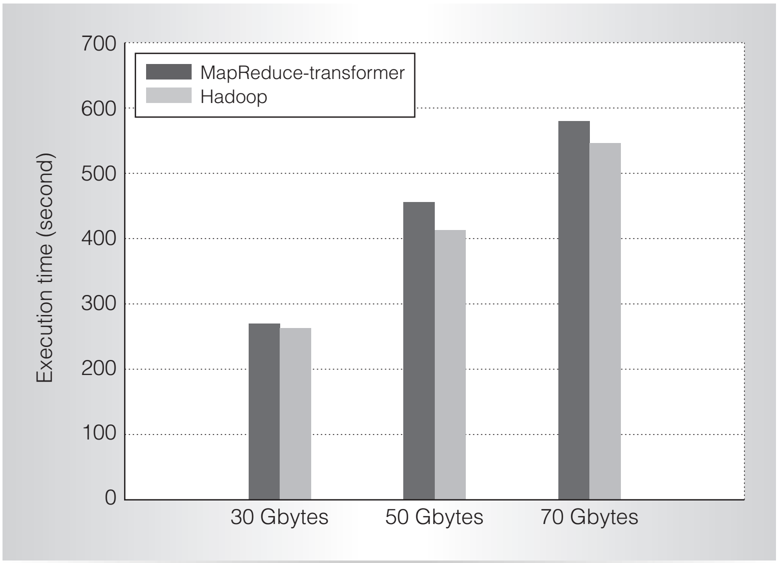

Figure 6. Experimental results from word-count application. Columns represent the execution time of Hadoop and MapReduce-transformer. Lower is better.

Data-flow evaluation

We compared our data-flow implementation with Hadoop using a real-life clickstream analysis, a large-scale data-intensive application running on a production system (more than 4 Tbytes per day) at Tencent.

On a website, clickstream analysis involves collecting, analyzing, and reporting aggregate data regarding how many visitors have visited, what pages they visited, and in what order they visited those pages. Clickstream analysis includes various statistical metrics. We only calculated three statistical metrics in this experiment: page views, user views, and analysis of source URL.

We used one real-world data set (collected by webservers) and two synthesis data sets for our experiment. We collected the log files from two sources: one from http://news.qq.com on 9 November 2009 (40.5 Gbytes total, 48 files) and one from http://qzone.qq.com on 22 December 2009 (36.95 Gbytes total, 48 files). Thus, the initial data size was 77.47 Gbytes (96 files). Data was stored as plain text; each line recorded a single row's information with columns separated by commas. We generated two other synthesis data sets by making three copies (232.4 Gbytes total) and five copies (386.9 Gbytes total) of the real-world data set, respectively. We stored all initial input data and final output data in HDFS. We stored the intermediate files in local disks and uploaded the final output to HDFS using Hadoop's command-line utility, as the final result size was small. Figure 5 shows the performance results in seconds of execution time. An important reason why Transformer's data flow outperformed Hadoop is that data flow achieves better data reduction than MapReduce. The performance results show the benefits of the data-flow model over the MapReduce model for some classes of workloads (for example, clickstream application).

MapReduce evaluation

We also made apples-to-apples comparisons between Hadoop and Transformer's MapReduce using a word-count application on different dataset sizes. Figure 6 shows the performance results. Our current implementation is a bit slower than Hadoop, mainly because we use a coarse-grained data-partition strategy for the map stage, whereas the input data in Hadoop is split into independent chunks (typically 64 Mbytes) that are read and processed individually by each map task. The fine granularity

Related work in scalable software infrastructure

Recently, driven by huge increases in dataset sizes, we've witnessed the development of various scalable software infrastructures for massive data processing. We believe that Transformer's unique features, such as a loosely coupled architecture, high-level abstraction (task execution and data transport), two concise primitives, actor-based implementation, and unified inter-component message passing make it different from existing systems in the data-intensive computing field.

Cluster-level infrastructure is an indispensable system for cluster resource management.[1] Hindman et al. provide an approach to multiplexing cluster resources among different existing programming frameworks such as Hadoop and Dryad.[2] Our target is narrower than those efforts and focuses on how to facilitate building diverse programming models.

Transformer is also related to previous work in distributed computation. Neither the `send()` nor `receive()` primitives are new, and the message passing interface (MPI, http://www.open-mpi.org) provides sophisticated communications support. However, our work proves the power of a simple communication set. In Transformer, we demonstrate how to use two primitives to build three different programming models.

The current form of Transformer (including our data-flow and MapReduce implementations) still lacks some important features of mature MapReduce/Dryad implementations, such as a flexible data-partition strategy, task progress monitoring, a backup task mechanism, and a fault recovery policy. It is, nevertheless, useful for investigating and prototyping new data-parallel programming systems.

References

1. L.A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines,* Morgan & Claypool, 2009.
2. B. Hindman et al., ''A Common Substrate for Cluster Computing,'' *Proc. HotCloud Workshop Hot Topics in Cloud Computing,* Usenix Assoc. Press, 2009, pp. 91-95.

tasks in Hadoop could improve dynamic load balancing and speed up recovery. With reference to our implementation, programmers could further improve performance by adopting a fine-grained data partition strategy in the map stage that exploits data locality.

We believe our system can enable a new paradigm for building programming models, useful to both researchers and Internet service companies. Researchers can apply our paradigm to exploring new programming models for massive data processing, or even extend the research beyond data-intensive computing. With reference to our implementation, Internet service companies can build a stack of general-purpose, robust, and scalable programming models. MICRO

Acknowledgments

This research was supported by Tencent Corporation. We thank Chuan Cai, Xiaochun Cui, Jun Liao, and Yinghui Liu for their support. We are grateful to the members of Tencent's Data Platform Department for their helpful feedback. We are also grateful to Luiz André Barroso, Partha Ranganathan, and the anonymous reviewers for their many invaluable comments and suggestions.

References

1. J. Dean and S. Ghemawat, ''MapReduce: Simplified Data Processing on Large Clusters,'' *Proc. 6th Conf. Symp. Operating Systems Design & Implementation,* Usenix Assoc. Press, vol. 6, 2004, pp. 137-150.
2. M. Isard et al., ''Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,'' *ACM SIGOPS Operating Systems Rev.,* vol. 41, no. 3, 2007, pp. 59-72.
3. B. Hindman et al., ''A Common Substrate for Cluster Computing,'' *Proc. HotCloud Workshop Hot Topics in Cloud Computing,* Usenix Assoc. Press, 2009, pp. 91-95.
4. C. Moretti et al., ''All-Pairs: An Abstraction for Data-Intensive Cloud Computing,'' *Proc. Int'l Parallel and Distributed Processing Symp.,* IEEE Press, 2008, pp. 1-11.
5. G. Malewicz et al., ''Pregel: A System for Large-Scale Graph Processing,'' *Proc. 28th ACM Symp. Principles of Distributed Computing,* ACM Press, 2009, p. 6.
6. R. Pike et al., ''Interpreting the Data: Parallel Analysis with Sawzall,'' *Scientific Programming,* vol. 13, no. 4, 2005, pp. 277-298.
7. C. Olston et al., ''Pig Latin: A Not-So-Foreign Language for Data Processing,'' *Proc. 2008 ACM SIGMOD Int'l Conf. Management of Data,* ACM Press, 2008, pp. 1099-1110.
8. A. Thusoo et al., ''Hive—Warehousing Solution over a Map-Reduce Framework,'' *Proc.*

Int'l Conf. Very Large Data Bases (VLDB), vol. 2, no. 2, VLDB Endowment, 2009, pp. 1626-1629.

9. R. Chaiken et al., ''Scope: Easy and Efficient Parallel Processing of Massive Datasets,'' *Proc. Int'l Conf. Very Large Data Bases (VLDB),* vol. 1, no. 2, VLDB Endowment, 2008, pp. 1265-1276.
10. Y. Yu et al., ''DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language,'' *Proc. 8th Symp. Operating Systems Design and Implementation,* Usenix Assoc. Press, 2008, pp. 1-14.
11. D. A. Patterson, ''Technical Perspective: The Data Center is the Computer,'' *Comm. ACM,* vol. 51, no. 1, 2008, p. 105.
12. L.A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines,* Morgan & Claypool Publishers, 2009.
13. W.M. Johnston, J.R.P. Hanna, and R.J. Millar, ''Advances in Dataflow Programming Languages,'' *ACM Computing Surveys,* vol. 36, no. 1, 2004, pp. 1-34.
14. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, 1986.

Peng Wang is a PhD student at the Institute of Computing Technology, Chinese Academy of Sciences, where his research focuses on programming models for massive data processing. Wang has an MS in computer science from the Ocean University of China.

Dan Meng is a professor of computer science at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include data-intensive computing and virtualization. Meng has a PhD in computer science from the Harbin Institute of Technology.

Jizhong Han is an associate professor of computer science at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include parallel processing and storage system. Han has a PhD in computer science from the Institute of Computing Technology.

Jianfeng Zhan is an associate researcher at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include system software for high-performance computing and energy-efficient computing systems. Zhan has a PhD in computer science from the Institute of Software, Chinese Academy of Sciences.

Bibo Tu is an assistant researcher at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include multicore computing and data-intensive computing. Tu has a PhD in computer science from the Institute of Computing Technology.

Xiaofeng Shi is a senior software engineer at Tencent Corporation, where he leads the team working on distributed computing platforms. His research interests include data-intensive computing and distributed file systems. Shi has an MS in computer science from Xian Jiaotong University.

Le Wan is a software engineer at Tencent Corporation, where he leads the development of clickstream applications. His research interests include data-intensive computing and data management. Wan has an MS in computer science from Jilin University.

Direct questions and comments about this article to Peng Wang, Institute of Computing Technology, Chinese Academy of Sciences, P.O. Box 2704, Beijing, China, 100190; wangpeng@ncic.ac.cn.