A Framework for an In-depth Comparison of Scale-up and Scale-out

Michael Sevilla, Ike Nassi, Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn
University of California, Santa Cruz
1156 High Street, Santa Cruz, CA
{msevilla, inassi, kleoni, scott, carlosm}@soe.ucsc.edu

ABSTRACT

When data grows too large, we scale to larger systems, either by scaling out or up. It is understood that scale-out and scale-up have different complexities and bottlenecks but a thorough comparison of the two architectures is challenging because of the diversity of their programming interfaces, their significantly different system environments, and their sensitivity to workload specifics. In this paper, we propose a novel comparison framework based on MapReduce that accounts for the application, its requirements, and its input size by considering input, software, and hardware parameters. Part of this framework requires implementing scale-out properties on scale-up and we discuss the complex trade-offs, interactions, and dependencies of these properties for two specific case studies (word count and sort). This work lays the foundation for future work in quantifying design decisions and in building a system that automatically compares architectures and selects the best one.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, Design, Performance, Standardization

1. INTRODUCTION

A critical part of designing a data-intensive computing system is choosing the best scaling[1] architecture for the targeted workloads. The two most popular scaling architectures are scale-out (which adds nodes to a system) and scale-up (which adds resources to a single node). Despite its well-documented limitations, Hadoop [7], the open source implementation of the popular scale-out MapReduce [5] framework, is still used for big data analytics because it is easy to program and the framework automatically distributes work to many interchangeable nodes; the ability to seamlessly add nodes gives scale-out the ability to, in theory, scale indefinitely.

Table 1: The word count application achieves different properties (indicated by a ✓) depending on the scaling architecture it is implemented on. The Hadoop version provides properties that are ignored in scale-up versions.

	scale-out	scale-up	
	Hadoop	Sequential	Parallelized
parallelism	✓		✓
fault tolerance	✓		
portability	✓	✓	
scalable storage	✓		
availability	✓	✓	
scalability	✓		

Many studies contest the notion of automatically choosing scale-out and make the convincing arguments that most "big data" working sets are not actually that big. For example, [12] shows that many of these working sets can fit in big memory systems and that scale-up performs better and costs less than its scale-out counterpart, even for "embarrassingly" parallel workloads. While we agree that scale-up has many performance benefits for today's "big data" jobs, we argue that there are also certain costs that need to be considered when choosing scale-up as the computation framework.

In addition to potentially forfeiting "indefinite scalability", a scale-up computation framework loses automatic parallelism, fault tolerance, portability, scalable data storage, and a high degree of availability. These properties influence the effectiveness of the system and may be required by the workload but often get overlooked in current research when scrapping for that last drop of performance [13]. A brief analysis of the word count application, shown in Table 1, illustrates the problem; a generic scale-out framework, like Hadoop, offers many nice properties but could be erroneously dismissed in favor of a specialized scale-up implementation purely based on performance measurements.

To choose the best scaling architecture, the architect needs to understand the trade-offs of the implementation decisions and the dependencies between the properties the system provides. Any attempt at quantifying the cost or benefit of these trade-offs/dependencies must be based on a fair com-

[1]Our definition of scaling includes the addition of memory, processing power, and storage capacity and bandwidth.

DISCS-2013, November 18 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2534645.2534654.

parison. Such a comparison between scale-out and scale-up is difficult because of the diversity of their programming interfaces, their significantly different system environments, and their sensitivity to workload specifics. How can system architects define equivalence between such different architectures and what are the parameters to consider in comparisons? What hardware and software metrics, that account for the properties that are inherently ensured on scale-out, can be used to make the systems equivalent? Acknowledging these properties is not a new concept [6] but we use them in a scale-out/up comparison framework and make the following contributions:

1. A framework based on MapReduce for comparing scale-out and scale-up that encompasses the workload, its property requirements, and its input size (§3).
2. An analysis of the trade-offs we considered when achieving 3 scale-out properties on scale-up. The design decisions heavily influence these properties (§5).
3. An analysis of the dependencies amongst these properties. Achieving one heavily influences the effectiveness of another (§5).

Comparing the pure scalability of the two systems is a different topic that requires examining multiple physical layouts of both scale-out and scale-up and is beyond the scope of this paper. Instead, we focus on standard scale-up and scale-out system architectures commonly explored in the literature and highlight the effects of achieving parallelism, fault tolerance, and scalable storage on scale-up. This work lays the foundation for future work in quantifying design decisions and, ultimately, developing a system that selects either scale-out or scale-up given the expected workload.

2. PRELIMINARIES

To make this study feasible, we limit our definition of "scale-out" to the MapReduce programming model. This decision provides little opportunity for advancing parallel programming techniques or comparisons [4] but we focus on MapReduce for two reasons: (1) MapReduce has become the standard for big data analytics and (2) we want to make fair comparisons and this model has API/runtime implementations for single nodes. We define "scale-up" to be a single node architecture (one enclosure, power supply, etc.). Below, we present several properties that are inherent to scale-out which must be considered when comparing scale-up to scale-out.

2.1 Parallelism: scale-out leverages parallel programming concepts to automatically distribute and balance load across nodes.

2.2 Fault tolerance: scale-out has the ability to sustain crashes during a job - failures are the norm and computation is rescheduled when nodes fail.

2.3 Scalable storage: scale-out uses a distributed file system to store data; despite inefficiencies, it provides scalable storage and 3× replication.

2.4 Pure scalability: scale-out can circumvent resource limits by adding nodes without changing the runtime, API, or application (via configuration files).

2.5 Portability: scale-out Hadoop applications can run on any Hadoop cluster, which allows applications to move to systems with different hardware ratios.

2.6 Availability: scale-out can always service clients; unavailable (update, upgrade, failed, etc.) nodes affect performance, not the computation itself.

An examination of the first three properties is presented in this paper; the ability to scale, portability, and availability all play roles in choosing either scale-out or scale-up but these are topics for future exploration. An in-depth cost comparison is also outside the scope of this paper and has been analyzed in [12] and used as the only parameter in the comparison framework in [9].

3. METHODOLOGY

Quantifying the costs of the design decisions in scale-out and scale-up requires a framework for an in-depth and fair comparison. Such a comparison must ensure the two systems are equivalent, which is complicated because the systems have very different architectures and programming models with variables that depend on the architecture and the end user's goals. For example, how does the system architect quantify properties, like fault tolerance, that are provided by an architecture "for free"? How does this measurement change if the client or problem does not require fault tolerance? How do we ensure that the algorithms on the architectures are equivalent? Our comparison framework helps answer these questions and is constructed using important system parameters.

3.1 Comparison Parameters

To ensure that the two systems we are comparing are equivalent, we must identify the variables that affect performance.

Software parameters affect the application itself and they consist of (i) the problem, (ii) the algorithm to solve the problem, and (iii) the subset of the scale-out properties in §2 that the solution guarantees.

The **input parameters** are the type of workload and the input size to the solution.

The **hardware parameters** are the processing power of the system and the available memory. Other parameters that also affect performance, such as the network latencies, node compute speeds, and disk bandwidths, are treated as constants because they are dependent on hardware specifics.

3.2 Comparison Framework

Below, we present a system for designers to properly construct a fair comparison between scale-out and scale-up, given the expansive variable space.

3.2.1 Choosing software parameters

(i) The **problem**: we choose problems with solutions that exercise different system parameters: word count and sorting. Instead of implementing, tuning, and deploying a wide range of applications on scale-up and scale-out, like [2], we decided to fully examine the effects of these two applications.

MapReduce's word count represents an optimal parallelized implementation, since a small percentage of the execution time is sequential. The application filters a small set of words from a larger set and the small shuffle/output data sizes make the job CPU bound. For proof, [8] shows high CPU utilization during the map phases, which can be attributed to the 60GB input set producing a 4GB shuffle set and a 1.6GB output set. MapReduce's sort is IO bound [8]

and leverages scale-out's ability to utilize idle compute power and disk bandwidth on each node. In Hadoop, as the mappers and reducers are running, background processes are helping the computation by sorting data as keys are being written to disk after the map phase and as keys are being ingested by the reduce phase.

(ii) The **algorithm**: we pick "matching" algorithms to solve the given problem. To "match" the algorithms, we encompass a wide range of implementations by taking a scale-out implementation and porting to scale-up for both methodology (i.e. imitating the scale-out algorithm) and functionality (i.e. focusing on the end-goal without restricting the algorithm choice). This framework transparently exposes costs at both ends of the implementation spectrum in an effort to stress the differences between the architectures.

(iii) The **scale-out properties**: we combine the two ported solutions and add modules to achieve scale-out properties on scale-up.

3.2.2 Choosing input parameters

To study data-intensive computing, we fix the hardware and increase the data on every experiment. By scaling the input size, we force the workload to stress different aspects of the problem being solved. We use the same parameters to generate the random input for each experiment and take the average of three experiments.

3.2.3 Choosing hardware parameters

We define a compute context to be an entity that is assigned a unit of work; for scale-out, this is an entire node and for scale-up this is a thread. We statically match the compute contexts and available RAM of the two systems. This is a fair comparison because each compute context computes on the same sized unit of work. This static hardware partitioning represents the "cross-over point" where scale-up is no longer clearly better than scale-out. Repeating the experiments for multiple hardware platforms would attest to the pure scalability of the system but is beyond the scope of this work.

4. EXPERIMENTAL SETUP

To port for methodology, we use Phoenix [11, 15], an API/runtime that converts MapReduce programs to multicore, multiprocessor programs. Phoenix replaces MapReduce nodes with threads and network communication with shared-memory. Using Phoenix leverages the MapReduce programming model and algorithms. We consider using a single-node version of Hadoop to preserve the algorithm but removing all the "distributed systems" aspects from Hadoop is involved [2] and we feel that a system designed for many nodes has a limited ceiling on scale-up. To port for functionality we use sequential implementations because they are easy to reason about and implement.

Our scale-out system uses Hadoop, an open-source API/runtime implementation of MapReduce, and has 32 nodes, each with 8 GB of RAM and 2 dual-core processors (32 total hardware contexts). Despite the older hardware[2], our methodology and results are still valid because (1) the nodes are not stressed or overutilized because of how Hadoop is implemented, and (2) we maximize visibility into our scale-out system; cloud solutions like Amazon's EC2 platform may introduce subtle overheads because of virtualization, locality, etc. For simplicity, Hadoop is configured with the default settings (2 map tasks per job, 1 reduce task per job, and a 64MB HDFS block size). Our scale-up system is running Red Hat Enterprise Linux 6 and has 256 GB of RAM and 2 quad-core processors with hyper-threading enabled (32 hardware contexts).

[2]Our cluster is about 6 years old, so it has older memory bandwidths, CPU speeds, and cache sizes.

Table 2: Our comparison framework requires implementations of scale-out properties on scale-up. Below, we list the trade-offs of different implementation decisions (columns) and the ✓s indicate which implementation is better for which scale-out property (rows).

Legend: ph. = Phoenix, had. = Hadoop, dmc. = DMTCP

	parallelism		fault tolerance		scalable storage	
	ph.	had.	dmc.	xen	hdfs	swift
performance	✓		✓			✓
parallelism	✓	✓	✓		✓	✓
fault tolerance		✓	✓	✓	✓	✓
portability		✓		✓	✓	
scalable storage		✓	✓		✓	✓
availability		✓			✓	✓
our justification	§5.1	[2]	§5.2	§5.2	§5.3	[14]
our decision	✓		✓		✓	

5. ANALYSIS

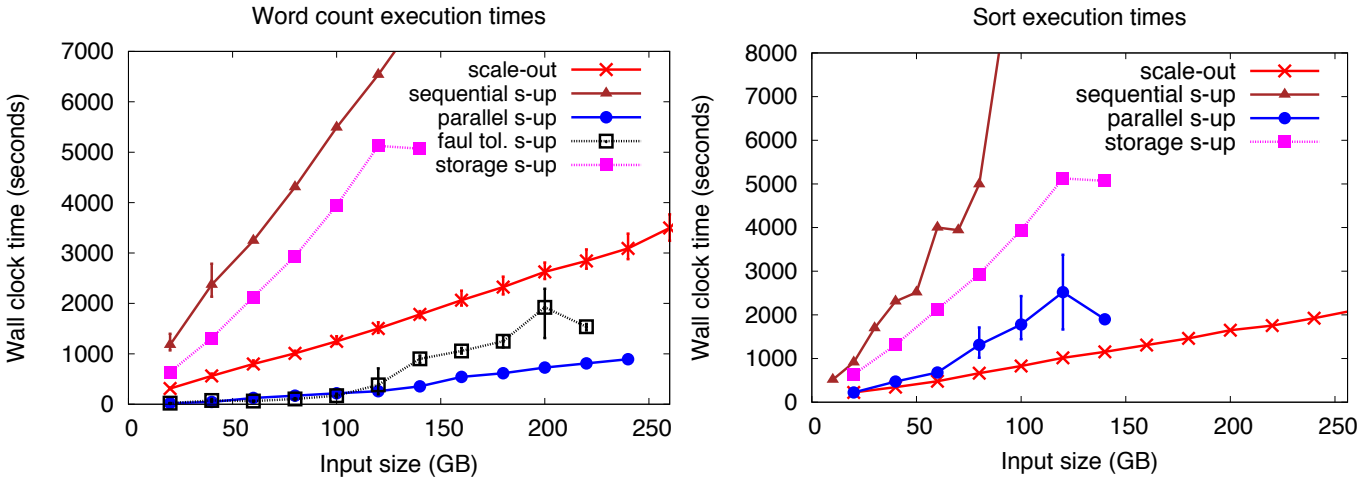
Table 2 details the implementation trade-offs and property dependencies we identify and we justify the claims we make when we discuss how we implement parallelism (§5.1), fault tolerance (§5.2), and scalable storage (§5.3). Figure 1 shows how achieving each scale-out property on scale-up affects performance for the word count and sort applications. The scale-out curve is the default Hadoop implementation, the parallelized scale-up curve is the Phoenix implementation, the sequential scale-up curve uses arrays to count or sort words, and the fault tolerance curve is the DMTCP implementation. For reference, the HDFS transfer speed is also plotted as the storage scale-up curve.

We do not discuss the performance results of our sequential implementations but they provide a reference for the effects of achieving the different properties, especially parallelism [13]. While distributed processing will usually outperform a sequential implementation, it is not unreasonable for distributed management overheads to dominate a parallel application for some workloads, leading to poor performance.

5.1 Trade-offs for Parallelism

To automatically parallelize programs, we use the Phoenix runtime, which comes with a word count implementation. We wrote a Phoenix sort application, which exercises the underlying framework to sort the data; the mappers and reducers output raw key-value pair without computation.

Performance is still limited by Amdahl's law because of the introduction of serial job phases. For some applications, this is acceptable because the benefit of the parallelized computation offsets the serial penalties. For example, Figure 1a shows the parallel scale-up word count implementation achieves a 3.4× speedup over scale-out at



(a) Word count **(b)** Sort

Figure 1: Because word count and sort output different amounts of key-value pairs, our framework recommends different scaling architectures. Taking into account different workload requirements (i.e. the properties) and input sizes exposes implementation trade-offs and property dependencies, which are presented in §5.

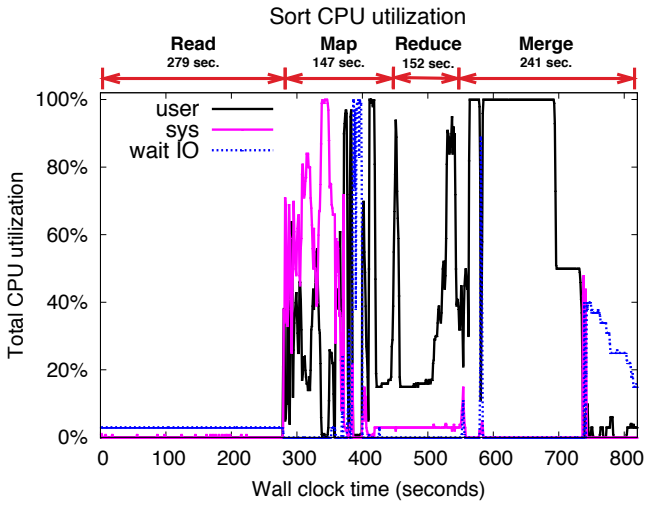


Figure 2: At a 60 GB input size, the CPU utilization for the Phoenix sort application exposes additional phases, which prove to the bottlenecks in the job. The IO wait is very low during the read phase because only one core (3% of the total CPU utilization) reads the data from disk into memory.

240 GB. The application is no longer CPU bound because it underutilizes the compute power of the scale-out system. Because we did not stress the system, the job is bound by I/O and network, which is directly related to the number of intermediate/final key value pairs.

On the other hand, parallelized applications that introduce long serial job phases on scale-up can experience worse performance than scale-out. Figure 1b shows the parallel scale-up sort implementation experiencing a 1.64× slowdown compared to Hadoop and large performance deviations at 120 GB input. Word count is faster because it has many repetitive words so the key-value pairs can be compressed; sort must process $\frac{n}{100}$ 100 byte key-value pairs at every stage.

Figure 2 shows the job's CPU utilization at the last stable input size, 60 GB. The y-axis is the percentage of time spent in kernel-space code (`sys`), user-space code (`user`), and in code waiting for IO (`IO wait`). The low utilization from 0 - 279 seconds and 700 - 800 seconds reveals bottlenecks in the job introduced by the Phoenix runtime; in this case, the intervals correspond to new read and merge phases, respectively. These long serial phases are circumvented in scale-out Hadoop with parallel reads from different nodes and by merging sorted data in background processes.

In an attempt to minimize the length of these phases, we used OpenMP [10], an API for shared memory multiprocessing, to get finer control of the job phases. Attempts at reading data into memory in parallel were limited by the disk bandwidth of our 3-drive RAID-0 device (384 MB/s maximum). On large enough RAID devices, parallelism could reduce this read phase significantly. We also implemented an OpenMP sort version that achieves better parallelism when merging data and avoids the useless reduce phase in the Phoenix implementation but the job is limited to one thread when parsing the key-value pairs; this computation is parallelized in Phoenix during the map phase.

Fault tolerance, scalable storage, and portability are not supported. Single node Hadoop achieves these properties with replication, HDFS, and heterogenous hardware but Phoenix and OpenMP forfeit these properties.

Availability is reduced if the degree of parallelism is not limited. The Phoenix word count and sort implementation achieve 100% CPU utilization during the computation and merge (shown in Figure 2) phase, respectively. This would block any incoming jobs from running until a core is released; a load balancing algorithm would be required to achieve the desired availability of the system.

5.2 Trade-offs for Fault Tolerance

To achieve fault tolerance, we use tools to checkpoint intermediate data to disk so that we can resume computation if the node crashes. Initially, we tried Xen's `xm save` to checkpoint the RAM state to disk but for large RAM

sizes, the overhead proved overwhelming because the checkpoint time took longer than the actual computation time. Instead, we use Distributed MultiThreaded CheckPointing (DMTCP) [1] to checkpoint an application's execution context to disk. DMTCP allows the application to restart from the last checkpoint, is application agnostic, and only checkpoints the application context.

Performance is dictated by the checkpoint interval; to reason about the trade-offs/dependencies of checkpointing, designers must understand the system characteristics and the application's parameters, such as the total computation time and rate, throughput of reading data into memory, and input size. Although the fault tolerant scale-up curve in Figure 1a shows a 1.85× speedup over scale-out in the worst case (240 GB), it is still 1.66× slower than a non-checkpointed version. At the 240 GB input size, while the data is being read into memory, DMTCP starts checkpointing, resulting in massive IO activity on 3 cores. We could use RAMdisk to reduce the checkpoint latency but RAMdisk does not scale well (writes scale poorly because of swapping) and would reduce portability (requires installation).

Changing the checkpoint granularity can lead to a shorter recovery time, however, as shown by the word count measurements in Figure 3, it comes at a performance cost. In the same figure, the scale-out curve is Hadoop (which inherently achieves fault tolerance), the "none" curve is the Phoenix word count implementation, and the DMTCP curve is the Phoenix word count implementation with DMTCP enabled at different checkpoint intervals. We also plot the Xen performance for a single checkpoint, for reference. For this 140 GB input size, a checkpoint interval of somewhere between 4 and 5 minutes makes the fault tolerant scale-up implementation slower than the Hadoop scale-out implementation.

The benefits of other scale-out properties are negatively affected. The degree of parallelism is reduced because checkpointing makes the job largely sequential. For example, for the fault tolerant Phoenix word count implementation, only 9% of the total job time is fully parallelized. Scalable storage performance and space is reduced because these large checkpoints require a significant amount of write bandwidth and capacity. The checkpoints depend on the size of the input - for our large experiments, this amounts to about 245 GB of data. Availability is reduced for the same reasons presented in §5.1. Portability is reduced since DMTCP is not widely used and although Xen and MPI checkpointing are popular, they are not "more portable" than ignoring checkpointing entirely.

5.3 Trade-offs for Scalable Storage

With data sizes easily reaching the petabyte range, it is unlikely that a single disk can accommodate "big data" data sets, so we evaluate a "scale-up computation; scale-out storage architecture". Scale-out storage for the cloud or large data centers falls into two main categories: object and block storage. Cloud storage architectures, like OpenStack's Swift [14], provide scalable object storage and can be used as a back end for computation frameworks like Hadoop. Distributed file systems provide block storage for large data sets and can be directly integrated with Hadoop. Despite their performance and architectural differences, both scalable storage solutions encounter the same bottleneck if they use a scale-up computation framework: getting the data from many nodes onto one node.

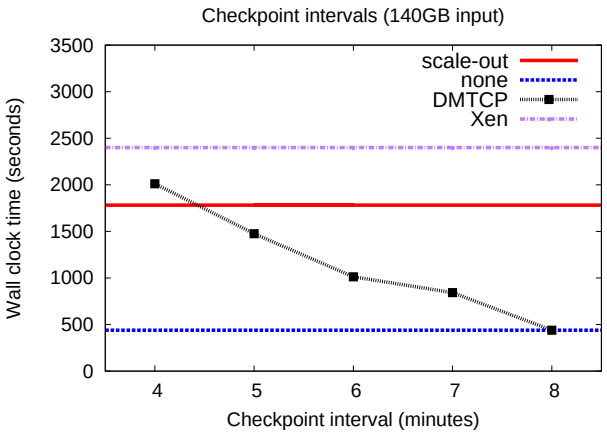


Figure 3: Checkpointing with a small interval improves granularity but could make the word count job slower than Hadoop. Note: the Xen curve is the time to perform one checkpoint, not a fault tolerant Xen word count.

Since all "scale-up computation; scale-out storage" architectures are limited to this bottleneck, we attempt to quantify a simple implementation. We use HDFS, which provides storage scalability and 3× replication, as the primary storage for one compute node and look at the transfer speeds of HDFS's `copyToLocal`.

Performance is limited by our hardware setup and the storage architecture. Our implementation moves large amounts of data through one 1 Gigabit network port. Furthermore, HDFS is not designed for this type of data transfer; it is designed for moving computation to the data. As a result, the storage overhead of "scale-up computation; scale-out storage" is overwhelming, as shown by the scalable storage curve in Figure 1. We could improve performance by adding more networking ports, upgrading the speed of our network, or by filtering irrelevant data.

Parallelism, fault tolerance, and portability are all inherited from the file system. HDFS is known to be bottlenecked by a single master node. We could improve parallelism and fault tolerance by using Ceph [17], which pushes responsibilities to object storage devices. We could get the same portability as Hadoop with OpenStack's new (unreleased) version, which will support direct integration of Swift and Hadoop.

6. RELATED WORK

Current work benchmarking Hadoop and optimizing parameters for Hadoop examine the effects of different workloads [8, 3]. Although these types of studies accurately characterize jobs by their resource dependencies, they do not discuss how these dependencies might change on scale-up. We leverage these results and accommodate the workload's property requirements and input size to make fair comparisons between scale-out and scale-up. Although early scale-out/up studies clearly enumerate the trade-offs between scale-out and scale-up, they use a simple comparison framework based on cost or are limited by out-dated hardware, methodologies, and benchmarks [9, 16].

Modern scale-out/up studies question the notion that we should scale-out by default and show that current scale-up

systems may suffice given the current composition of "big data" jobs and because of today's downward trending hardware prices [12, 2]. Although we agree with this idea[3], we contend that there are other scale-out properties to consider when comparing performance. We align more with the ideas presented in [13] - in fact, they have already identified two of our scale-out properties, although they do not provide experimental analysis. The most useful resources for these studies are blogs and whitepapers, many of which have already highlighted the importance of acknowledging scale-out properties when comparing to scale-up. For example, [6] advocates this technique and suggests automating the comparison process (a concept achieved in [2]). We agree with these concepts but classify them as future work for our comparison framework.

The community has made extensive contributions to parallel programming co-design [4] to determine which applications are good for specific scale-up architectures but we leverage the MapReduce model to compare scale-out/up algorithms and to achieve automatic parallelism. Work that advances parallel programming can help assist future iterations of our framework but in this work, we focus on comparing scale-up to scale-out.

7. CONCLUSIONS AND FUTURE WORK

We show that when judging MapReduce on scale-up, there are many other parameters that must be considered. We present a comparison framework that encompasses the input, software, and hardware parameters to make scale-out and scale-up MapReduce systems equivalent. Part of our framework requires achieving scale-out properties on scale-up and we show that system properties are tightly-coupled to each other and to their implementations. Parallelism affects and is affected by all other system properties, fault tolerance could make scale-up Hadoop slower than scale-out Hadoop, and scalable storage is a huge bottleneck without sufficient hardware.

Future work will be to quantify the trade-offs and dependencies in an attempt to develop automatic tools for deciding between different scaling architectures given a job, its property requirements, and its input size. We also intend to automate the comparison process so that the portability is more comparable to the models in [4] or concepts in [6, 2].

ACKNOWLEDGEMENTS

We would like to sincerely thank all of our anonymous reviewers for their helpful comments and suggestions, especially in regards to related works that we missed or future directions we should pursue.

8. REFERENCES

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th ACM Symposium on Cloud Computing*, 2013.

[3]We paid $9,000 for our scale-up system, which initially had 384 GB of RAM, and $2,000 for each scale-out node.

[3] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 137–142, 2010.

[4] K. Czechowski and R. Vuduc. A Theoretical Framework for Algorithm-architecture Co-design. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*, pages 791–802, 2013.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, 2004.

[6] Gigaspaces. Scale Up vs. Scale Out. In *Gigaspaces Resource Center*. http://www.gigaspaces.com/WhitePapers, 2011.

[7] Hadoop. http://hadoop.apache.org/, accessed 08/09/2012.

[8] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *Internatioanl Conference on Data Engineering Workshops*, pages 41–51, 2010.

[9] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2007.

[10] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, accessed 08/09/2012.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[12] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody Ever Got Fired for using Hadoop on a Cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pages 2:1–2:5, 2012.

[13] M. Schwarzkopf, D. G. Murray, and S. Hand. The Seven Deadly Sins of Cloud Computing Research. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, pages 1–1, 2012.

[14] Swift. http://www.openstack.org/software, accessed 08/09/2012.

[15] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-memory Systems. In *Proceedings of the 2nd International Workshop on MapReduce and its Applications*, pages 9–16, 2011.

[16] A. Talkington and K. Dixit. Scaling-Up or Out. *International Business*, 2002.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design & Implementation*, pages 307–320, 2006.