Masters Project Rough Draft: Quantifying the Difference Between Scale-Out and Up

Michael Sevilla
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
msevilla@soe.ucsc.edu

ABSTRACT

When data grows too large, we scale to larger systems, either by scaling out or scaling up. It is understood that scale-up and scale-out have different complexities and bottlenecks, but comparing them is difficult because the programming interfaces and environments are very different. In this vision paper, we present a complete framework for comparing these architectures and quantify the costs of achieving scale-out properties on scale-up. We then use this framework to show that a scale-up system that achieves scale-out properties can still outperform scale-out, even if the total compute power and workload favor scale-out.

1. INTRODUCTION

When data exceeds the processing or storage capacity of a system, the focus shifts to scaling to a bigger system, either by scaling out (adding nodes to a system) or scaling up (adding resources to a single node). For small data sets, scale-up architectures usually enjoy familiar, shared memory programming models and, if the data set can fit in memory, high performance. Unfortunately, the scalability of the system is dictated by its physical resources, as shown in Figure 1. We call the point where the system exhausts its resources the "resource wall[1]". When scale-up systems approach the resource wall, performance usually slows to a crawl. Scale-out (e.g., MapReduce [7]) scales better for large data sets because the framework distributes work to many *interchangeable* nodes; the ability to seamlessly add or remove nodes allows scale-out to circumvent the resource wall.

But scale-out architectures are starting to face challenges, resulting in workload-specific architectures, optimizations made at the application level, and incredible complexity/unpredictably at the system maintenance level (have you ever managed a cluster?). For examples of workload-specific architectures, look no further than the current literature; because MapReduce [7] confines the user to specific inputs and workloads, specialized systems like Pregel [16], Spark [37], and S4 [20], were developed to handle graph processing, iterative/interactive, and steam processing computations, respectively.

[1]Inspired by [35]'s "memory wall", which is a subset of the "resource wall".

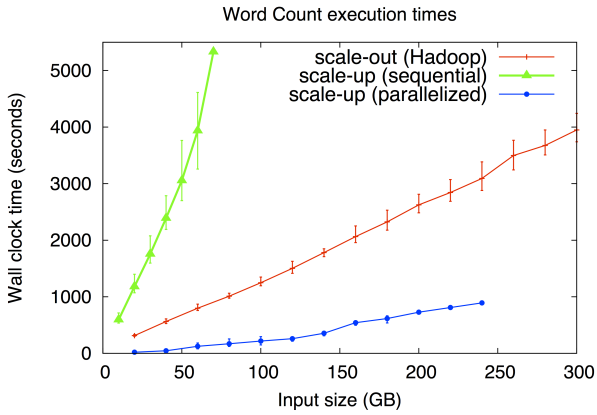


Figure 2: Scale-up (32 cores/256GB RAM) outperforms a much more powerful scale-out (128 cores/256GB RAM) system. This shows that, using existing solutions, scale-up can achieve better performance than scale-out *even if* (1) the scale-out system has more compute power and (2) the workload is "embarrassingly" parallel.

Choosing the best scaling approach has the potential to benefit the overall performance, cost, and flexibility of cloud applications, but in-depth comparisons between the architectures is hard because the systems are very different. In this paper, we provide an expansive framework that we use for an in-depth comparison of the two scaling approaches, which helps expose many tradeoffs in performance that can be made when dealing with big data. Previous work has been done to explore these tradeoffs and our work quantifies their costs.

Rowstron et al. [23] contest the notion of automatically

run	$\frac{\text{memory}}{\text{core}}$ Ratio	Cores, RAM
M_1	0.5	4, 8GB
M_2	1.5	8, 12GB
M_4	2.5	4, 10GB

→ sequential program: add words to array, sort array
→ parallel program: distribute work; use MapReduce algorithm

Our technique for modifying the core to memory ratios is to methodically increment ratios and maximize the resources. Varying the core to memory ratios gives different performance curves for increasing data. Curves with similar slopes and performance can help indicate an overprovisioning of resources. The results on the right show the "resource wall", which is the point at which performance degrades due to insufficient resources; in this case, main memory.

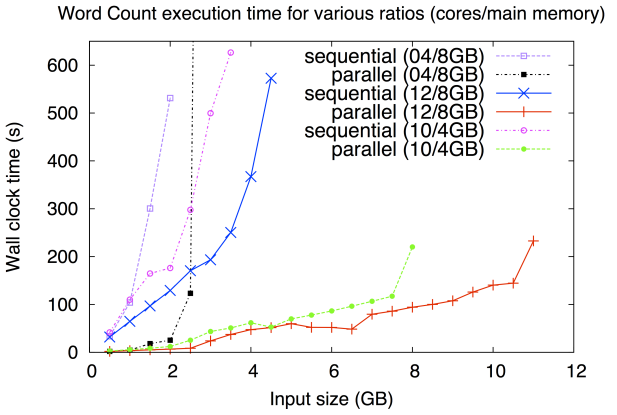


Figure 1: Single node experiments showing scale-up's inability to dynamically add resources and the effects of a limited resources.

choosing scale-out and they make the convincing arguments that (1) most working sets can fit inside big memory systems and (2) scale-up performs better and costs less than its scale-out counterpart, even for "embarrassingly" parallel workloads. These arguments are intuitive, since replacing slow/remote components in scale-out with fast/local components in scale-up benefits performance. We verify and experimentally quantify the difference in performance in Figure 2 and show that scale-up outperforms scale-out *even if* the workload and total resources favor scale-out.

Although scale-up has many benefits, there are also certain costs we need to consider when choosing scale-up. In addition to losing the flexible resource wall, we lose automatic parallelization, fault tolerance, portability, data storage/replication, and a high degree of availability (discussed in §2). While we agree that improving performance on current systems and finding unique ways to get around the resource wall is important, we want to call attention to other properties that affect performance - properties that often get overlooked in current research when scrapping for that last drop of performance [25]. We quantify the cost, in performance, of these scale-out properties on a scale-up system by making the following contributions:

1. We introduce a complete framework for comparing scale-out and scale-up using software that ensures we compare architectures, not application behavior (§3).
2. We achieve automatic parallelization, fault tolerance, portability, and data storage/replication on scale-up with low enough overheads, such that the system can still outperform scale-out (§4).
3. We show that achieving scale-out properties on scale-up change overall performance, resource utilization, and system behavior (§5).

2. PRELIMINARIES

We use the following definitions for simplicity and clarity. We limit our definition of "scale-out" to the MapReduce [7] programming model, which was designed for huge jobs and scalability, because it is the standard for big data analytics. We limit "scale-up" to single node architectures (one enclosure, power supply, programming image, etc.) for enterprise or academic settings. Note that we are targeting systems in the enterprise and academic communities. Work done in the business and high performance computing communities, like the Cray XK6 [11] and the IBM BlueGene/Q [12], are outside the scope of this paper because of the price range and scale.

Below, we present 7 scale-out properties that affect performance when implemented in a scale-up system. Scale-out, especially when deployed in the cloud, is ideal for workloads that need these properties, so we consider the cost of achieving them on scale-up when making comparisons.

1. **parallelism**: scale-out leverages parallel programming concepts to automatically (using map/reduce tasks) distribute and balance load across nodes.
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.
3. **portability**: scale-out Hadoop applications can run on any Hadoop cluster, which allows applications to move to systems with different hardware ratios.
4. more **storage/replication**: scale-out uses a distributed file system to store data; despite inefficiencies [27], it provides scalable storage and 3× replication.
5. **cheaper**: in the past, it was cheaper to scale-out because spreading RAM over commodity hardware was cheaper than consolidating RAM in one box.
6. **availability**: scale-out can always service clients; "down" (update, upgrade, failed, etc.) nodes affect performance, not the computation itself.

7. more **resources**: scale-out can circumvent resource limits by adding nodes without changing the runtime, API, or application (via configuration files).

3. METHODOLOGY

In our experiments, we compare our scale-out system against our single node scale-up system. Our scale-out system uses Hadoop, an open-source API/runtime implementation of MapReduce, and has 32 nodes, each with 8GB of RAM and 2 dual-core processors (4 hardware contexts). Our cluster is about 6 years old, so the nodes have out-dated memory bandwidth speeds, CPU speeds, and cache sizes. For reference, LMBench's [18] STREAM copy latency is 11472MB/s for our scale-up system and 2343 MB/s for one of our scale-out nodes.

Initially, Hadoop is configured with the default settings (2 map tasks per job, 1 reduce task per job, 64MB HDFS block size) and if performance is unreasonably slow, we tweak parameters. Note that this paper is not about Hadoop or its configuration - we just notice in our experiments that the system can produce unreasonable results if poorly configured. For example, Terasort with an 80GB input set took three hours with the default Hadoop configuration and 5 minutes when we increased the HDFS block size and the number of reducers.

Our scale-up system is running Red Hat Enterprise Linux 6 and has 256GB of RAM and 2 quad-core processors with hyperthreading (32 hardware contexts). We choose these systems because they have the same aggregate RAM. We do not match compute power (i.e. core counts) because this reduces our scale-out system to 8 nodes, which is unfair to Hadoop, a system designed for many nodes.

To properly study scalability, we fix memory size and the number of processors and increase the data. For each run, we continually add data in 20GB chunks until the application or operating system crashes, following the psuedo-code below:

```
foreach application
   while(!stressed)
      execute()
      measure_performance()
      ++data
```

For example, on the first iteration, we will process a file of size 20GB, then a file of size 40GB, then a file of size 60GB, and so on. After a fresh reboot, each experiment is run once without profiling and stat collecting to warm up the OS and hardware caches and experiments are run three times.

3.1 Comparing Architectures

It is challenging to specify application equivalence between such radically different architectures, so we encompass a wide range of implementations by taking a scale-out implementation and porting it to scale-up for both methodology (i.e. the algorithm) and functionality (i.e. the end goal). This framework transparently exposes costs at both ends of the implementation spectrum in an effort to stress the differences between the *architectures* instead of the *applications*.

	MapReduce	Phoenix
work distr.	master node worker nodes	parent process threads ∈ core
communication	network i-keys ∈ HDFS	shared-memory i-keys ∈ L1 cache
fault tolerance	heartbeat remote re-execute	timeout local re-execute
combiner	∈ node after map	∈ thread after map
API	`string`	`void *`

Figure 3: The differences between Phoenix and MapReduce. Phoenix is an open-source implementation of MapReduce for shared-memory systems.

To port for methodology, we use the Phoenix systems [22, 36, 31], which are a set of APIs/runtimes that convert MapReduce programs to multicore, multiprocessor systems. Phoenix replaces MapReduce nodes with threads and network communication with shared-memory, essentially leveraging MapReduce's data parallelism and functional programming model. A component by component breakdown is shown in Figure 3.

To port for functionality we use the best and most intuitive solution for the given architecture. For example, if the task is to sort, we would compare a sequential algorithm (like Quicksort) on scale-up to a distributed systems algorithm (like Hadoop's sort) on scale-out.

3.2 Monitoring and Profiling

We use a thorough methodology for gathering information about the scale-up system's behavior when it accommodates scale-out properties. We argue that achieving scale-out properties on scale-up has an appreciable effect on:

- **latencies**, which changes execution time.
- **utilization**, which changes bottlenecks and slow downs.
- **hardware efficiency**, which changes system behavior.

We collect this information with system wide profilers, available on Linux. We use the `time` program to analyze latencies and the proportion of time spent in each binary (i.e. application, library, or operating system code). We use the System Activity Report (SAR) program to analyze resource utilization for the CPU, main memory, and swap space on disk. Finally, we use OProfile to statistically sample the hardware performance counters, to collect statistics for cache activities and CPU utilization ratios.

3.3 Workloads

Our workloads are the embarrassingly parallel word count, sort, and K-means clustering applications because we want to show that, even for the best workload, scale-out can outperform scale-up. For example, only 1% of the word count program is sequential [15], making it ideal for Hadoop - in fact, many tutorials and papers use the word count example to explain how MapReduce works.

Word count counts the number of occurrences of distinct words in an input set, sort orders randomly generated 100-byte keys, and K-means clustering iteratively groups numerical n-dimensional vectors based on their mean-squared distances. [14] classifies the distributed Hadoop versions of each of these benchmarks: word count is CPU bound because it aggressively combines values (reduces intermediate key-value pairs and network traffic), sort is I/O and network bound because it produces a large amount of non-compressed intermediate key-value pairs, and K-means is CPU-bound during the map base and I/O bound during the cluster phase.

Our sequential implementations do not use any Phoenix calls, but they use the same system calls and parsing techniques as their parallelized equivalents. Word count and sort iterate over the input file, store each pair in an array, and use `qsort` to sort the output by value. The K-means program generates random points and means and iterates over each point (one at a time) to identify the clusters. Our parallel implementations are discussed in 4.1.

The input for the parallel word count and sort benchmarks are generated with the `randomtextwriter` and `teragen` Hadoop modules (in the example `.jars`), respectively. On our large single node, we run a single node instance of Hadoop to generate the data, and then copy the data to our local drive using `copyToLocal`. On our scale-out system, we use the same module to generate the same amount of data. In this way, the input data is produced in *exactly* the same way. The input for K-means is randomly generated at runtime.

4. IMPLEMENTATION

We achieve following properties on scale-up to show to help quantify the differences between the architectures. Table 1, provides a summary of the tradeoffs we consider and indicates our final design decisions. Note that all sections could list "performance" as a tradeoff, but again, we want to show that the design decisions affect other properties, in addition to "performance". In the rest of this section, we discuss each property in depth and explain how we achieve it.

Recall that properties 6 and 7 are not in the scope of our experimental setup but could be achieved with a replicated node. Achieving scale-out property 5, with an in-depth cost comparison, is beyond the scope of this paper, but it is worth noting that cost may not be a benefit of scale-out any longer. Current work on scaling architectures [23, 3] show that scale-up memory prices are trending to affordable when compared to scale-out systems; for example, we paid $9000 for our scale-up system and $2000 for each node in our scale-out system.

4.1 Achieving parallelism on scale-up

Parallelism is usually achieved on scale-up using parallel programming techniques. Unfortunately, parallel programming is difficult [17] because the programmer has to deal with concurrency (synchronization, messages, locks, etc.), resources (shared-memory, thread/process, locality, etc.), applications (re-tuning, portability, scalability, etc.), and significantly larger code bases.

We use Phoenix to achieve parallelism in the same way that a MapReduce programmer does. Each MapReduce algorithm is structured around the same data flow. The input is split along boundaries, each mapper emits an intermediate key-value pair (`<key, 1>`, `<key, value>`, and `<min index, point>` for word count, sort, and K-means, respectively), and a combiner combines intermediate results. Each reducer handles specific key ranges; reducers pull the intermediate data from the correct location and compute the total value for each key. Word count aggregates words on pre-defined reducers, sort lets the framework sort the data by using identity mappers/reducers, and K-means clusters points in parallel.

Phoenix++ has a word count and K-means implementation in its distribution and we wrote our own sort program using the Phoenix APIs. Phoenix enables us to use the MapReduce programming model on scale-up by defining the splitter, partitioner, mapper, and reducer functions. The Phoenix API also supports special containers for optimizing key organization and location. As an aside, we benchmark the Hadoop sort algorithm against Hadoop's Terasort because the Terasort input is easier to parse and verify; the Hadoop sort program sorts a sequence file and the output is bytes that do not appear to have any order. The only difference between the two algorithms is that Terasort uses a custom partitioner, which builds a trie from sample keys to optimize key placement on certain reducers.

4.2 Achieving fault tolerance on scale-up

Fault tolerance is the ability to sustain crashes during a computation or job. On a scale-out system, when a node crashes, the system takes a performance hit but recovers and completes the job. For example, in MapReduce, failures are accepted as the norm and when a node dies, computation is rescheduled on another node.

Initially, we used Xen's `xm save` to checkpoint a virtual machine image because it only takes 2 minutes on a regular sized machine - but on a system with 256GB of RAM, the checkpoint took about 40 minutes, which is longer than the computation itself, as shown in Figure 4. Xen also has trouble utilizing a hypervisor with too much memory so we had to waste resources and limit the usable memory by changing the bootloader configurations.

To achieve fault tolerance, we use Distributed MultiThreaded CheckPointing (DMTCP) [2] to checkpoint and restart an application's execution context. DMTCP checkpoints a specified application, including its memory, open files, and partial results, to disk. We checkpoint every 5 minutes so that we can recover in-flight computation for any external (power, tornado, earthquake, etc.) or internal (CPU throttling, disk corruption, etc.) error. This solution allows the application to restart from the last checkpoint, is application agnostic, and **only** checkpoints the application context.

4.3 Achieving portability on scale-up

Scale-up programming (especially for C++) is not nearly as portable as scale-out, since programmers write code to leverage hardware characteristics. For example, the second version of Phoenix allows the user to set environment variables and the running application queries for hardware configurations, such as cache size and number of processors using

Table 1: Tradeoffs and dependencies of achieving scale-out properties on scale-up. The design decisions we made are marked with a ✓. To read the table, say:
"If we <Design Decision> with <cell value>, we get better <col.header>!"
— e.g. "If we read input with `mmap()`, we get better performance!"

	Design Decision	Performance	Scalability	Portability	Storage space	Granularity	We have it (i.e. Cost)
‖-ism	read input	✓ mmap()*	malloc()†				✓ software
	store int. <k,v>	✓ hash		array			✓ software
	implement ‖-ism	✓ Phoenix		Hadoop	Hadoop		✓ software
fault tol.	use compression	✓ none			gzip∓		✓ software
	use intervals	i=30min			i=30min	✓ i=5min	✓ software
	store checkpoints	RAMDisk	✓ HDD•	✓ HDD⋆			✓ software
	restart	RAMDisk	✓ HDD	✓ HDD			✓ software
storage	stripe data	RAID0‡	RAID0				✓ 1 HDD
	store data	SSD$^{\gamma}$	SSD				✓ HDD
	use network ports	multiple$^{\flat}$	multiple				✓ single
	use Hadoop	Rhea$^{\alpha}$	Rhea	✓ default	Rhea		✓ default
	use a file system	Ceph$^{\beta}$	Ceph	✓ HDFS			✓ HDFS

- Notes -

*in Phoenix 2, the authors note that `malloc()` performance is limited by `read()`

†in Phoenix 2, `mmap()` addressable range limited to 2GB (`int` parameter)

∓we notice checkpoints being as large as the input, up to 256GB in our tests

•RAMdisk writes scale poorly because swapping is common

⋆RAMdisk requires installation

‡can perform I/O in parallel, achieving roughly $n\times$ performance (n drives)

$^{\flat}$HDFSs parallel copy (distcpp) can utilize multiple ports

$^{\alpha}$Rhea [13] can filter input sets to reduce network traffic

$^{\beta}$Ceph [33] pushes computation to OSDs (no reliance on single master node)

$^{\gamma}$see [23] for more details

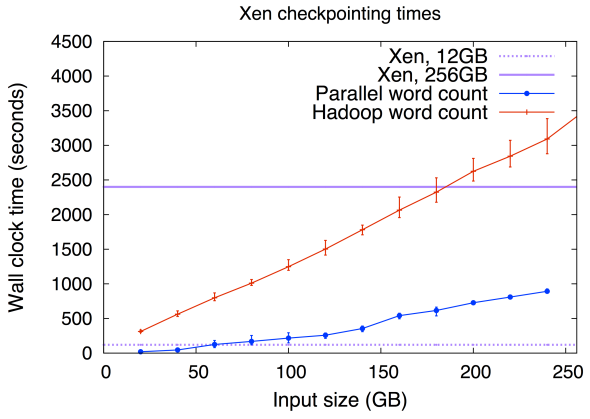


Figure 4: Xen checkpointing is too slow to provide fault tolerance for word count on large memory machines.

`getenv()`. If no environment variables are found, the runtime resorts to hardcoded values. This makes the application specific to the architecture and ultimately non-portable.

As a proof of concept, we implement a C library and bash script that queries the hardware and sets environment variables so that an application can use these values to align its data structures. Our library uses the `lscpu` Linux program to collect CPU information and it is structured as shown in Figure 5.

Tradeoffs: we implement two versions of the `setenvs` program: a static C library and a dynamic bash script. In our implementation, we use a static library, since the developer already knows the optimal data structure sizes as a function of a specific hardware feature. The bash script is useful for testing various data structure sizes.

4.4 Achieving storage/replication on scale-up

Using a single node limits the system to a local storage system, making scalable storage and replication more complicated. RAID schemes, most notably RAID5, are used extensively in large systems, and they can achieve replication with parity, but big data sets are quickly outpacing current storage solutions; it is becoming common to reach petabytes in scale (with exabytes on the horizon) and, at this time, we cannot store a petabyte of data in one machine.

We quantify the overhead of using a "scale-up computation; scale-out storage" architecture, which has one compute node

```
/* Application calls setenvs_static()
 * to query hardware
 * inp1: 'envar' is the environment variable
 *    to set, e.g., "MR_L1CACHESIZE"
 * inp2: 'component' is the hardware spec.
 *    to query e.g., "L1d cache"
 */
setenvs_static(envar, component)
   // Write output of 'lscpu' to a file
   execvp("lscpu") >> '.lscpu.tmp'

   // Iterate through file to find cache sizes
   foreach line in '.lscpu.tmp'
      value = extract component from line

   // Set the environment variable
   setenv(envar, value)
...

/* Inside the word_count.c application code*/
main()
...
   setenvs_static("L1d cache", "MR_L1CACHESIZE")
   setenvs_static("CPU(s)", "MR_NUMTHREADS")
...
```

Figure 5: Pseudo-code for the `setenvs_static.c` library, which queries the hardware and allows the applications to tune their data structures for hardware specifications.

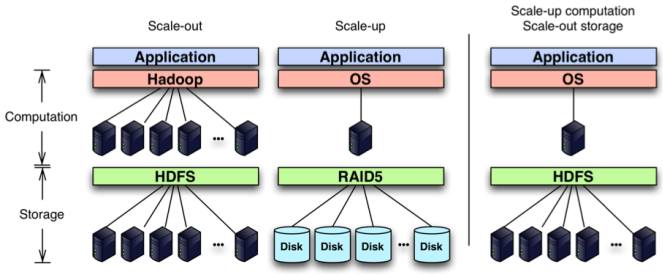


Figure 6: The "scale-out"/"scale-up" schemes are existing storage solutions; we evaluate the "scale-up computation; scale-out storage" architecture, which leverages the performance of scale-up and the storage/replication of scale-out.

sitting on top of an HDFS cluster, as shown in Figure 6. In this setup, HDFS provides storage scalability and 3× replication for the data and the compute node does all the computation.

To examine the costs of using a hybrid store-compute system, with one powerful compute node and many small slave storage nodes, we look at HDFS's internal `copyToLocal` program. We copy different sized inputs to the local harddrive of a single node and measure its performance. Note that this is a crude and naive test, but it helps illustrate how achieving this property on scale-up may not be trivial.

5. ANALYSIS

The costs of achieving scale-out properties 1 through 4 are shown in Figure 7. The figure will be referred to in each analysis subsection. From our analysis, we learn that achieving:

1. parallelism and portability can benefit performance
2. fault tolerance and data replication/storage has performance costs
3. scale-out properties affect utilizations, which changes bottlenecks and resource walls
4. scale-out properties affect hardware efficiency, which changes software behavior

5.1 Benefits of parallelism

In all three benchmarks, parallelizing the application with Phoenix improves performance over its sequential counterpart, as shown in Figure 7. Parellelization also has the potential to scale better than Hadoop for some applications; word count is consistently better but sort gets unpredictable because there are many more key-value pairs. Hadoop scales better for sort because it partitions data and compresses intermediate key-value pairs; if Phoenix had a method for implementing Terasort's partitioning trie or a way to compress intermediate key-value pairs, we might achieve better performance. All benchmarks still encounter the resource wall (we notice CPU throttling and segmentation faults when memory mapping the file) and performance drops off quickly when swapping to disk, but sufficient memory gives ideal results.

We quantify the benefits of parallelism in Figure 8a; the CPU time (`user`) is higher than the wall clock time (`real`), since the time spent on each core is aggregated. This shows that the job is being completed faster than the sequential job because each core is doing more work.

Porting our applications to a single node also changes the application behavior so drastically that they are no longer "bounded" by the same resource. For example, [14] identifies the distributed Hadoop word count as CPU-bound and K-means as CPU-bound in the map phase and disk-bound in the cluster phase. But Figure 8 shows that both parallel and sequential word count are CPU and disk-bound and that K-means is purely CPU-bound! In the period before 145 seconds we can tell that parallel word count is reading in data from disk because our tests shows the CPU waiting for I/O. Our implementation of sequential sort is bounded by a

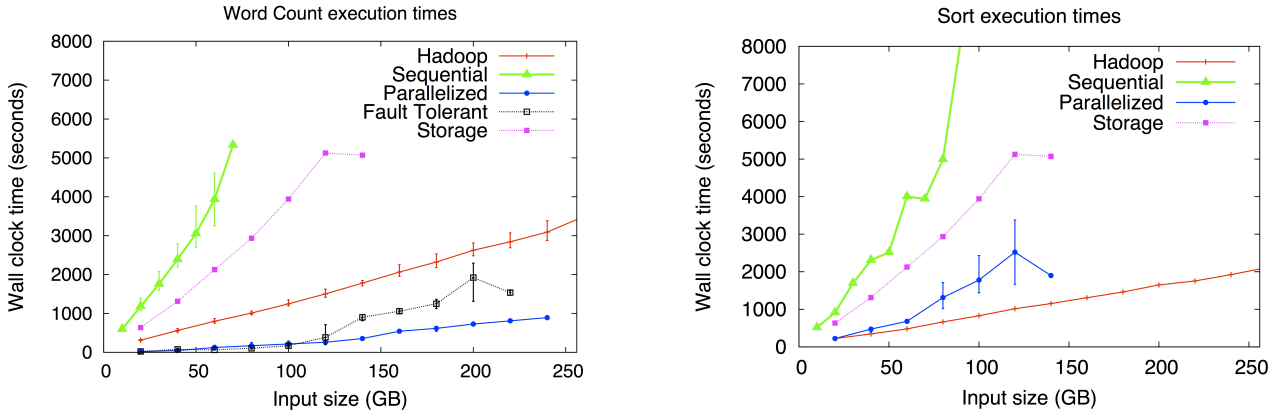
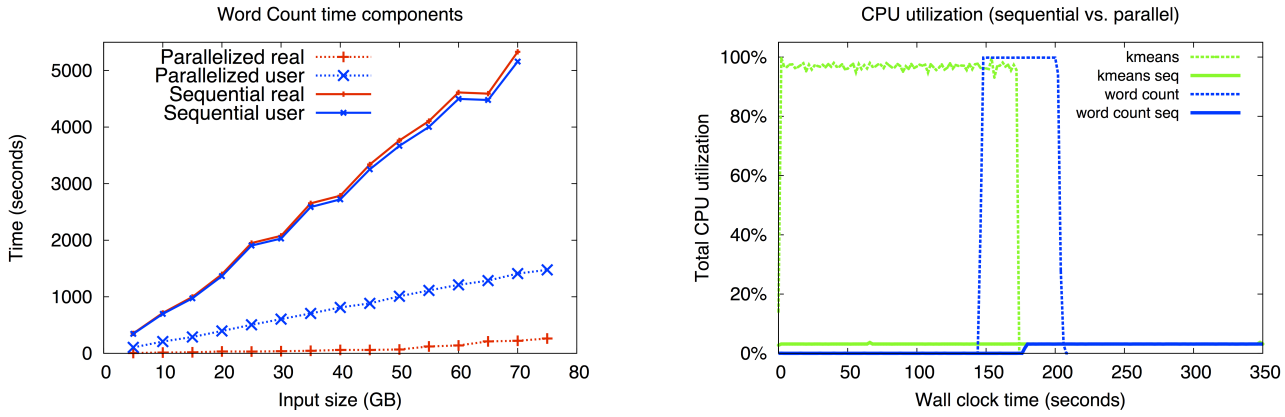


Figure 7: To quantify the costs of achieving scale-out properties on scale-up, we compare the performance of sequential and distributed (Hadoop) applications to the same applications with parallelism, fault tolerance, and unlimited storage using Phoenix (§5.1), DMTCP (§5.2), and HDFS (§5.4), respectively. Phoenix sort performs poorly because of the large amount of intermediate key-value pairs and the lack of partitioning and compression.



(a) **`real`** is the wall clock time and **`user`** is the amount of CPU time spent in the application.

(b) Parallelization makes K-means purely CPU bound and sequential word count disk bound.

Figure 8: Achieving parallelism benefits performance and changes resource utilizations.

single CPU; in our results, we see excessive CPU time spent waiting for I/O, leading us to believe sequential word count is bounded by the disk because a single thread sequentially ingests the data. K-means is no longer disk-bound because the application does not output intermediate key-value pairs to disk since everything is kept in memory.

5.2 Costs of fault tolerance

Achieving fault tolerance has a cost in performance and alters the system's resource utilization and behavior. Ultimately, this forces the application to lose some of the benefits of parallelism and portability because it is bottlenecked by the compute power of one core and the disk I/O.

Figure 7 shows the performance degradation for achieving fault tolerance; we see that in word count, the overhead starts at 125GB. Our checkpoint interval is every 5 minutes, so we start to a see a difference when the computation time starts to take longer than 5 minutes. Despite the overhead, performance scales well, even when checkpointing multiple times. For the sort benchmark, DMTCP is unable to checkpoint the computation because there are too many intermediate keys. The latency of an individual checkpoint is unknown, since we have not found a way to differentiate the transactions in DMTCP.

The resource utilization changes when accommodating fault tolerance because checkpointing is (1) limited to one core and (2) bottlenecked by disk I/O. Figure 9a shows the CPU utilizations for word count with and without fault tolerance. `user` is the time spent in user-space code and `sys` is the time spent in the operating system. Both word count versions have a memory mapping phase (400 - 700 seconds) and a computation phase (`user` spikes to 100%), but in the fault tolerant word count version, the file system is stressed as state is written to disk. As a result, the job execution time is dominated by time spent in the operating system. Also, in the fault tolerant version, a majority of the CPU time is spent on one CPU (two hyper threads amount to 6.25% of the total compute power), instead of distributed across many cores.

Achieving fault tolerance also increases swap activity, despite the same memory usage patterns. Figure 9b shows the memory utilization and swap activity for word count with and without fault tolerance. With checkpointing, the operating system must use more memory so that it can checkpoint the word count application - since the word count application already exhausted all the free space, our checkpointing application has to swap some of the memory out to disk to complete its own job.

Finally, we show that a poorly configured fault tolerant implementation has the potential to destroy an application's performance. Figure 9c shows the system thrashing because the job continually tries to checkpoint too much data (in this case, many key value pairs). At each checkpoint interval, the system "runs out of time" and cannot complete its checkpoint, so it spends all of its time context switching between the operating system and the application.

5.3 Benefits of portability

Table 2: Word count performance counters

		image		symbol	
default	L1D miss	word_count	64%	`emit_inter`	45%
		libc	34%	`strcmp()`	29%
	L1D evicted	libc	33%	`strcmp()`	29%
	CPU cycles	libc	26%	`strcmp()`	22%
portability	L1D miss	word_count	61%	`emit_inter`	44%
		libc	37%	`strcmp()`	34%
	L1D evicted	libc	37%	`strcmp()`	34%
	CPU cycles	libc	29%	`strcmp()`	26%

Portability benefits performance but it changes the hardware efficiency and different binaries start using the hardware in different ways. As a proof of concept, we implement a C library and bash script that queries the hardware and sets environment variables so that when an application runs, it can align its data structures to the returned hardware specifications.

The performance results (Figure 10) show a modest 2 second speedup for the Phoenix 2 word count test after the cache size was queried and scaled by 10; this is a result of using hardware specifications (like cache sizes) more efficiently in the application code. The details are hidden by the Phoenix API/runtime, but according to the TUNING documentation, the cache size is set to the "data chunk size" and each map task operates on a single data chunk. What we see in our results is the library forcing a bigger chunk size leading to a higher computation to communication ratio, at the cost of load balancing.

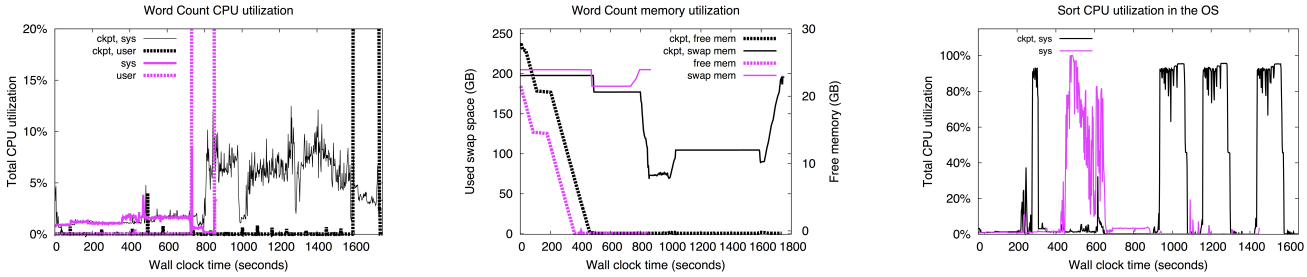
The hardware efficiency statistics show a change in system behavior, as different binaries start to utilize the hardware differently. When using our portability libraries, we observe an overall decrease in L1 data cache misses (about 650 samples) and Table 2 shows that portability affects where cache misses occur. Less of the misses occur in the word count application and more occur in the C libraries. Although overall hardware efficiency improves, the C libraries start to stress the system more than the application itself.

We would like to reiterate the real contribution, which is the portability library. It allows the same application to remain optimized if it migrates to different systems. In essence, the application developer has the power to manually tune the application for hardware specifications.

Tradeoffs: we implement two versions of the `setenvs` program: a static C library and a dynamic bash script. In our implementation, we use a static library, since the developer already knows the optimal data structure sizes as a function of a specific hardware feature. The bash script is useful for testing various data structure sizes.

5.4 Costs of storage/replication

The storage overhead of "scale-up computation; scale-out storage" in Figure 7 seems to be overwhelming because it is far slower than just computing on the data and because it scales poorly. When using the system to move data, we are



(a) The CPU activity shows that word count's behavior changes if we achieve fault tolerance.

(b) The memory activity shows that word count swaps frequently if we achieve fault tolerance.

(c) The CPU activity shows that sort thrashes in the operating system if we achieve fault tolerance.

Figure 9: Analysis of the effects of fault tolerance for word count and sort.

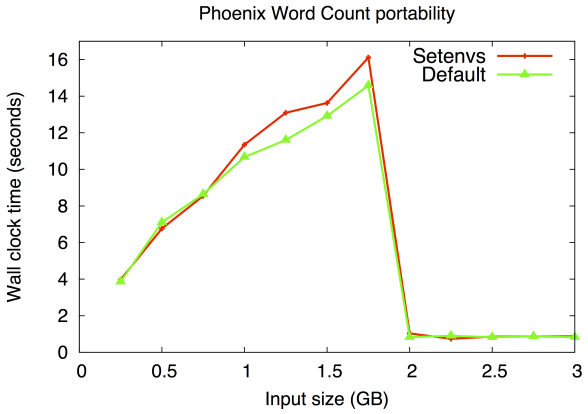


Figure 10: Portability: using our libraries, we achieve optimal application configuration automatically. At compile time, the operating system is queried and application environment variables are instantiated.

bottlenecked by the single node characteristics (e.g. file system, RAID configuration, storage medium, network ports, etc.). Also, Hadoop is designed for moving computation amongst nodes, so when the system moves a lot of data, it gets bottlenecked by the single node.

Despite this, we are confident that the "scale-up computation, scale-out storage" architecture is feasible, if we apply the techniques in Table 1.

6. RELATED WORK

There is extensive literature for comparing scale-up and scale-out in both the academic and non-academic communities. Published material is rather thin, usually using narrow methodologies and resorting to small benchmarks.

6.1 Directly comparing scale-out and scale-out

Previous scalability studies [32, 19] enumerate the tradeoffs between scale-out and scale-up but the studies use out-dated hardware, methodologies, and benchmarks.

Michael et. al. [19] present a direct comparison of scale-out and scale-up by configuring equivalent cost machines ($200,000) for a narrow workload. They determined that scale-out has a resource utilization cost despite a performance advantage, which, for their workload, was up to a 4× speedup. Interestingly enough "scale-out-in-a-box" has better performance than pure scale-up.

In a similar study, Talkington et. al. [32] observed the behavior of a powerful SMP system with scientific (SPEC), commercial (TPC-H), and compute-intensive (Fluent) benchmarks. The study concluded that scale-up is limited by processor and memory speeds and resource contention while scale-out is limited by intra-node communication and workload management. More specifically, scale-up shows diminishing returns at 32 processors for the given workloads.

These studies succinctly enumerate the differences between the scaling approaches but they are limited by a number of factors. First, the studies are out-dated, and as a consequence, the benchmarks, cost, hardware specifications, and system configurations are no longer applicable. Second, the studies are relatively shallow. The benchmarks are limited and the systems are highly tuned resulting in performance

measurements that make sense. It is our opinion that a more in-depth study with modern hardware, benchmarks, and system configurations will yield different results.

Modern scalability studies question the notion that we should "scale-out" by default [23, 3] and make a compelling case for moving computation to scale-up. Although we agree with this idea, we contend that there are other scale-out properties to consider when comparing performance. We align more with the ideas presented in [25] - in fact, two of our scale-out properties are already listed in [25].

6.2 Scale-up: Examining System Architecture

The arrival of experimental operating systems signifies an exploration of the system design space to accommodate increasing data and hardware trends. Barrelfish [24] proposes a new OS architecture which facilitates heterogenous hardware and application demands by maintaing a system knowledge base and making execution decisions at runtime. FOS [34] proposes a new OS architecture which distributes kernel system services spatially instead of time-sharing them. Corey [4] addresses multiprocessor scalability by giving applications control of shared kernel data structures. The system introduces address ranges, kernel cores, and user-space sharing via system calls. Cerberus [29] is based on the observation that commodity operating systems cannot scale for a large number of CPU cores. The system introduces OS-clustering to provide a single OS image (via a SuperProcess) to the user while virtualizing resources with Xen (share address space, file system, file contents, NICs) via shared memory and a static resource allocation (i.e. 8 cores per OS). These systems all attack new workloads and data sizes by changing how the system internals are structured and organized.

In contrast to a system design overhaul, another avenue of exploration for accommodating more data is to better understand and fix current systems. The common approach for these kinds of studies is to vary workloads and system parameters, profile and trace to characterize the behavior, and to fix any observed bottlenecks. Boyd-Wickizer et. al. [5] iteratively added cores, identified Linux bottlenecks at the 48 core threshold, and fixed micro/macro system level problems. [6] examines individual system calls and presents functions they call, how often they are called, and how long they take. [1, 21, 26, 8, 9, 10] all present tools to help the programmer "see" what is going on at the lower levels.

These studies all make contributions towards increasing visibility into the system from the application level but they are not applied to scale-out vs. scale-up studies. We leverage these tools and techniques to a get a better feel for how the system is behaving and to quantify what are causing various slowdowns in both architectures.

The push for new system designs and experimental operating systems, such as [24, 34, 29], show the community's willingness to explore a new design space for increasing data and modern hardware trends. Another way to accommosdate more data is to better understand, using profiling and tracing techniques, and fix current systems [5, 6].

The parallel programming community has extensive contributions to parallelism, like determinism [15], but they are complicated and do not leverage the MapReduce model. There is also excellent work to provide scale-up fault tolerance through the OS, like file system work in [30, 38], or through application space, like redundant processes [28], but these address specific system faults and correctness; DMTCP allows restart from any application at any time. Portability is not a new concept, for example, Java is "portable" because it is highly specified, but our implementation achieves Hadoop-like portability on scale-up. Finally, our work does not leverage distributed storage system research, but file systems like [33] have the potential to greatly improve performance of our "scale-out compute; scale-up storage" architecture.

7. CONCLUSION

Scale-out and scale-up have different properties that affect performance. In this paper, we argue for a more thorough evaluation of scaling approaches and we present a complete framework for comparing scale-out and scale-up, which helps quantify the costs of the tradeoffs. We also show that a scale-up system with scale-out properties can outperform scale-out, *even if* the compute power and workload favors scale-out. We analyze the affects of achieving scale-out properties on scale-up and provide evidence of how they affect latencies, utilization, and hardware efficiency. This leads to new performance measurements, bottlenecks, and system behaviors. This study raises more questions than answers and each component can be a thesis on its own, but the focus of this paper was to show performance costs of the decisions we make.

The last two scale-out properties, higher availability and more resources, are left as future work. It is not immediately obvious how to achieve these properties on a *single node* scale-up system, but what if we scale-up one system image with many nodes? Future work will look at systems that can virtualize at a layer lower than the application level. Such a system should have the ability to add nodes, thus mitigating the costs of providing availability and additional resources.

Acknowledgments

Special thanks to Kleoni and Ike for helping me with this project.

8. REFERENCES

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of the linux kernel. In *In Proceedings of the Real-Time Technology and Applications Symposium (RTAS02)*, page 133, 2002.

[2] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[3] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Nobody ever got fired for buying a cluster. Technical report, Mcirosoft Research, Cambrdige, UK, February 2013.

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[5] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[6] S. S. P. G. Bridges and A. B. Maccabe. A framework for analyzing linux system overheads on hpc applications. In *Proceedings of the 2005 Los Alamos Computer Science Institute (LACSI '05)*, page 17, 2005.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operarting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[8] M. Desnoyers and M. R. Dagenais. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium 2008*, page 101, July 2008.

[9] M. Desnoyers, J. Desfossez, and D. Goulet. Lttng 2.0: Tracing for power users and developers - part 1, April 2012.

[10] M. Desnoyers, J. Desfossez, and D. Goulet. Lttng 2.0: Tracing for power users and developers - part 2, April 2012.

[11] M. Feldman. Cray unveils its first gpu supercomputer, May 2011.

[12] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[13] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron. Rhea: automatic filtering for unstructured cloud storage. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 343–356, Berkeley, CA, USA, 2013. USENIX Association.

[14] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*, pages 41–51, 2010.

[15] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[17] J. L. Manferdelli, N. K. Govindaraju, and C. Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, 2008.

[18] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[19] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[20] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[21] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[23] 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*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[24] A. Scheupbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[25] 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 Ccomputing*, HotCloud'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[26] S. Shende. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop 2, USENIX*, June 1999.

[27] K. o. V. Shvachko. HDFS Scalab ility: The Limits to Growth. *login: The Magazine of USENIX*, 35(2):6–16, Apr. 2010.

[28] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306. IEEE, 2007.

[29] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A

case for scaling applications to many-core with os clustering. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 61–76, New York, NY, USA, 2011. ACM.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM SIGOPS Operating Systems Review*, 37(5):207–222, 2003.

[31] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[32] A. Talkington and K. Dixit. Scaling-up or out. *International Business*, 2002.

[33] 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 and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[34] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[35] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[36] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[38] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60. USENIX Association, 2006.